



LE LANGAGE C++ MASTER I LES CLASSES

Jean-Baptiste.Yunes@univ-paris-diderot.fr
U.F.R. d'Informatique
Université Paris Diderot - Paris 7

- La classe comme définition d'un nouveau type concret :
 - au même titre que la construction `struct` du C

```
class MaClasse {  
    ...  
};
```

`MaClasse` est un nouveau type. On peut désormais déclarer des variables de ce type

```
int unEntier; // un nouvel entier (définition)  
MaClasse unObjet; // un nouvel objet à moi...
```

ce qui permet d'obtenir des objets (que l'on qualifiera de « construits statiquement »)

La classe comme moule de construction

- Puisque c'est un type, il faut lui associer des valeurs et des opérations (attributs - comportements) :
- Les éléments constitutifs de la déclaration d'une classe (comme type tels que nous les envisageons jusqu'à maintenant) sont :
 - des champs :
 - variables d'instance, *i.e.* qui seront propres à chaque objet de cette classe
 - des méthodes
 - les opérations à appliquer aux objets de la classe (des fonctions attachées aux objets)

L'accès à ces éléments doit être contrôlé (voir plus loin)

- Le concept élémentaire de compte en banque pourrait revêtir la forme suivante :

```
class CompteEnBanque {  
    int solde;  
public: // à voir plus tard...mais important!  
    int getSolde();  
    void deposer(int somme);  
    void retirer(int somme);  
};
```

Cette déclaration suffit à imaginer une utilisation possible d'un compte en banque :

```
CompteEnBanque monCompte;  
  
cout << "Reste " << monCompte.getSolde() << endl;  
monCompte.deposer(100);  
monCompte.retirer(23);
```

- La définition de la classe :
 - pour obtenir une définition, il faut définir les méthodes (fournir le code!)

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

Syntaxe : utilisation de l'opérateur de portée ::
pour « désigner » la méthode

- Convention de structure du code source :
 - la déclaration d'une classe `MaClasse` dans un fichier `MaClasse.hpp` OU `MaClasse.H`
 - la définition de la classe (des méthodes de) dans un fichier `MaClasse.cpp` OU `MaClasse.C`
 - l'utilisation de la classe nécessite l'import du fichier de déclaration par `#include`

Autres conventions : `.hxx`, `.cc`, `.cxx`

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

CompteEnBanque.hpp

```
#include "CompteEnBanque.hpp"
void CompteEnBanque::deposer(int s) {
    solde += s;
}
void CompteEnBanque::retirer(int s) {
    solde -= s;
}
int CompteEnBanque::getSolde() {
    return solde;
}
```

CompteEnBanque.cpp

```
#include <iostream>
using namespace std;
#include "CompteEnBanque.hpp"
```

```
int main() {
    CompteEnBanque unCompte;

    cout << unCompte.getSolde() << endl;
    unCompte.deposer(100);
    cout << unCompte.getSolde() << endl;
    return 0;
}
```

CompteEnBanqueTest.cpp

- Compilation séparée :

- via Makefile :

```
OBJS = CompteEnBanque.o CompteEnBanqueTest.o
.PHONY : all clean
```

```
all : CompteEnBanqueTest
```

```
CompteEnBanqueTest : $(OBJS)
    $(CXX) -o $@ $(OBJS)
```

```
clean :
    rm -f $(OBJS) CompteEnBanqueTest
```

Makefile

- l'exécution produit :

```
[Trotinette:~/tmp] yunes% ./CompteEnBanqueTest
4096
4196
[Trotinette:~/tmp] yunes%
```

Problème d'initialisation!

L'INITIALISATION LA CONSTRUCTION

- L'initialisation des objets :
- Pourquoi faire porter le chapeau à l'utilisateur ? Si un objet est mal initialisé c'est parce qu'à sa construction son fabriquant s'y est mal pris!

Si vous commandez une Delahaye rouge et qu'on vous la livre non peinte ou bleue vous êtes en droit de vous plaindre au fabricant...



- Pourquoi n'est-ce pas le cas en programmation ???

- L'initialisation des objets :
 - Ce n'est pas résoudre le problème que de proposer que la classe `CompteEnBanque` fournisse un service, une action, une méthode permettant de mettre le compte à zéro...

L'utilisateur peut ne pas le mettre à zéro (par ce qu'il ne le sait pas, ou par ce qu'il a oublié volontairement ou non), ou il peut tenter de le mettre à zéro à différents moments (inadéquats).

- L'initialisation des objets :
 - On propose alors de contrôler l'initialisation des objets (rappel l'initialisation ne se produit qu'une fois dans la vie d'une variable : à sa création) :
 - En obligeant l'utilisateur à fournir des valeurs le permettant
 - En C++, il existe la possibilité de définir des méthodes très spéciales permettant d'initialiser un objet : les constructeurs

- Les constructeurs C++ :
 - méthodes qui
 - portent le nom de la classe,
 - ne déclarent rien à renvoyer (pas **void**! RIEN)
 - sont appelées automatiquement après la création de l'objet (appel implicite)
 - ne peuvent être appelées explicitement...

```
class CompteEnBanque {  
private:  
    int solde;  
public:  
    CompteEnBanque(); // Un constructeur  
};  
  
CompteEnBanque::CompteEnBanque() {  
    solde = 0;  
}  
  
int main() {  
    CompteEnBanque monPremierCompte;  
}
```


- Il est fréquent et ordinaire que plusieurs constructeurs soient définis. Chacun d'eux correspond pour l'utilisateur à la façon de *commander* un objet.

```
class Ordinateur {  
    private:  
        long mem;  
        long disk;  
    public:  
        Ordinateur(long tailleMemoire, long tailleDisque);  
        Ordinateur();  
};  
  
Ordinateur::Ordinateur() {  
    mem = 1<<30; // 1 Gio (la norme indique 1 gibioctet)  
    disk = 40<<30; // 40 Gio  
}  
  
Ordinateur::Ordinateur(long m, long d) {  
    mem = m;  
    disk = d;  
}
```

- Il est fréquent que plusieurs constructeurs soient définis. Chacun d'eux correspond pour l'utilisateur à la façon de *commander* un objet.

```
#include "Ordinateur.hpp"

int main() {
    Ordinateur uneBecaneDeBase;
    Ordinateur uneSuperBecane(100<<30, 1<<40);
    Ordinateur uneBecane(1<<30); // interdit!
}
```

- Les responsabilités sont clairement définies :
 - **L'utilisateur** de la classe est obligé de se conformer aux scénarios de construction d'un objet : il est obligé de spécifier des valeurs comme indiqué dans l'un des constructeurs disponibles, si cela ne construit pas un objet correct ce n'est pas de sa faute!
 - En cas de mauvais comportement de l'objet, il peut se plaindre au fabriquant (bien identifié) qui ne peut se défaire de sa responsabilité...

- Les responsabilités sont clairement définies :
 - **Le concepteur** est responsable de la bonne initialisation des attributs d'un objet, de sorte que celui-ci se comporte ensuite correctement. La cohérence des données est de sa responsabilité. Sa difficulté est d'imaginer les différentes manières de vouloir construire un objet.
 - il doit adopter une démarche *user-centric* : qu'est ce que l'utilisateur peut raisonnablement fournir comme données pour initialiser un objet ?

- De l'uniformité syntaxique des initialisations...
- Rappel : pour les types primitifs, on a

```
int i=4;    // l'initialisation habituelle  
int j(4);   // l'initialisation comme fonction
```

- Pour les objets on a :

```
MaClasse c(12,"Bonjour",Math::sin(Math::PI));
```

- Si pour une classe donnée il existe un constructeur à un argument alors les syntaxes autorisées sont:

```
MyClass o(12);    // l'initialisation standard vu comme fonction  
MyClass o = 12;   // l'initialisation comme les types primitifs...
```


- Les objets sont parfois employés comme interface d'accès à des ressources externes à eux-mêmes. Ce qui sous-entend qu'un tel objet détient une référence à cet objet mais ne le contient pas. Attention on parle d'implémentation pas de conception ici!

Par exemple : un objet pourrait représenter une session de connexion à une base de données et ainsi abstraite la véritable base de données associée. Un autre objet pourrait détenir une zone mémoire allouée dynamiquement

- Les constructeurs sont alors employés pour initier/initialiser cette relation

- Par exemple : définissons une classe d'objet permettant de représenter une pile (oublions l'aspect dynamique lié à la taille)

```
class Pile {  
private:  
    int *stock;  
    int tailleMax, sommet;  
public:  
    Pile(int tailleMax);  
};  
  
Pile::Pile(int tm) {  
    stock = new int[tm];  
    tailleMax = tm;  
    sommet = 0;  
}
```

LA DESTRUCTION



- Désormais se pose la question de la disparition de tels objets. Que devient la relation établie ?
- Le langage C++ offre un moyen de réaliser des opérations particulières lorsqu'un objet disparaît :
 - Ce mécanisme s'appelle destructeur

- Un destructeur est :
 - Une méthode dont le nom est celui de la classe préfixée par le caractère ~
 - Une méthode ne déclarant rien à renvoyer (rien pas `void`) et ne prenant pas de paramètre
 - Une méthode automatiquement appelée à la destruction d'un objet. Cette destruction est implicite dans le cas d'un objet statiquement alloué ou explicite lorsqu'il s'agit de détruire un objet alloué dynamiquement
 - Une méthode qui ne peut être surchargée... Un seul destructeur par classe

- Pour notre pile cela pourrait être :

```
class Pile {
private:
    int *stock;
    int tailleMax, sommet;
public:
    Pile(int tailleMax);
    ~Pile();
};

Pile::Pile(int tm) {
    stock = new int[tm];
    tailleMax = tm;
    sommet = 0;
}

Pile::~~Pile() { // plus de fuite mémoire!!!!
    delete [] stock;
}
```

- Des constructions et destructions inattendues?

```
class Gloup {  
public:  
    Gloup();  
    ~Gloup();  
};
```

```
#include <iostream>  
using namespace std;  
#include "Gloup.hpp"  
  
Gloup::Gloup() {  
    cout << "Construction d'un gloup" << endl;  
}  
Gloup::~~Gloup() {  
    cout << "Destruction d'un gloup" << endl;  
}
```

- Des constructions et destructions inattendues ?

```
#include "Gloup.hpp"

void f(Gloup g1, Gloup g2) {
    Gloup *pg = new Gloup;
    delete pg;
}

int main() {
    Gloup g1, g2;
    f(g1, g2);
}
```

```
[Trotinette:~/tmp] yunes% ./Gloup
Construction d'un gloup
Construction d'un gloup
Construction d'un gloup
Destruction d'un gloup
Destruction d'un gloup
Destruction d'un gloup
Destruction d'un gloup
Destruction d'un gloup
[Trotinette:~/tmp]
```

Quoi ? 3 ctor et 5 dtor ?

- Rappel : le passage d'argument du C++ est comme celui du C

Passage par VALEUR

- À l'entrée d'une fonction recevant un argument par valeur, on **créé** une nouvelle variable du type considéré et dont le nom est celui du paramètre formel, variable qui est alors **initialisée** à l'aide de la valeur du paramètre effectif



- Dans notre exemple... On crée deux objets Gloup à l'entrée de la fonction, mais on ne voit pas d'appel au constructeur !!! Alors ?
- Parce qu'on crée une **variable initialisée** à la valeur du paramètre effectif

```
#include "Gloup.hpp"

void f(Gloup g1, Gloup g2) {
    Gloup *pg = new Gloup;
    delete pg;
}

int main() {
    Gloup g1, g2;
    f(g1, g2);
}
```

```
[Trotinette:~/tmp] yunes% ./Gloup
Construction d'un gloup
Construction d'un gloup
Construction d'un gloup
Destruction d'un gloup
Destruction d'un gloup
Destruction d'un gloup
Destruction d'un gloup
Destruction d'un
gloup[Trotinette:~/tmp]
```


- Il existe donc une notion de construction d'un objet de type T à l'aide d'un objet de type T (une sorte de clonage).
- Ce mécanisme est appelé **construction par copie**.

```
#include "Gloup.hpp"

void f(Gloup g1, Gloup g2) {
    Gloup *pg = new Gloup;
    delete pg;
}

int main() {
    Gloup g1, g2;
    f(g1, g2);
}
```

```
[Trotinette:~/tmp] yunes% ./Gloup
Construction d'un gloup
Construction d'un gloup
Construction d'un gloup
Destruction d'un gloup
Destruction d'un gloup
Destruction d'un gloup
Destruction d'un gloup
Destruction d'un gloup
[Trotinette:~/tmp]
```

- On peut imaginer la signature d'un constructeur par copie...

```
class Gloup {  
public:  
    Gloup();  
    Gloup(Gloup);  
    ~Gloup();  
};
```

- Mais on se mord la queue!!!! Le passage d'argument est par valeur aussi donc il faudrait pour appeler le constructeur par copie pour faire la copie, donc appeler la copie pour faire la copie permettant de faire la copie, donc...

- La bonne signature est :

```
class Gloup {  
public:  
    Gloup();  
    Gloup(const Gloup &);  
    ~Gloup();  
};
```

- Il s'agit d'un nouveau mode de passage de paramètre, appelé le passage par référence! On ne crée pas de nouvelle variable mais on désigne dans la fonction le paramètre effectif à l'aide d'un autre nom (comme un alias)

- Des constructions et destructions inattendues ?

```
class Gloup {  
public:  
    Gloup();  
    Gloup(const Gloup &);  
    ~Gloup();  
};
```

```
#include <iostream>  
using namespace std;  
#include "Gloup.hpp"  
Gloup::Gloup() {  
    cout << "Construction d'un gloup" << endl;  
}  
Gloup::Gloup(const Gloup &aCloner) {  
    cout << "Construction d'un gloup (par copie)" << endl;  
}  
Gloup::~~Gloup() {  
    cout << "Destruction d'un gloup" << endl;  
}
```

- Cette fois obtient :

```
#include "Gloup.hpp"
```

```
void f(Gloup g1, Gloup g2) {  
    Gloup *pg = new Gloup;  
    delete pg;  
}
```

```
int main() {  
    Gloup g1, g2;  
    f(g1, g2);  
}
```

```
[Trotinette:~/tmp] yunes% ./Gloup  
Construction d'un gloup  
Construction d'un gloup  
Construction d'un gloup (par copie)  
Construction d'un gloup (par copie)  
Construction d'un gloup  
Destruction d'un gloup  
Destruction d'un gloup  
Destruction d'un gloup  
Destruction d'un gloup  
Destruction d'un gloup  
[Trotinette:~/tmp] yunes%
```

- Il faut absolument noter l'existence de destructions implicites : lorsqu'on sort de la fonction `f` et lorsqu'on sort du `main`

- Le cas des valeurs de retour...

```
#include "Gloup.hpp"
```

```
Gloup f(Gloup g1, Gloup g2) {  
    Gloup *pg = new Gloup;  
    delete pg;  
    return g1;  
}
```

```
int main() {  
    Gloup g1, g2;  
    g1 = f(g1, g2);  
}
```

```
[Trotinette:~/tmp] yunes% ./Gloup  
Construction d'un gloup  
Construction d'un gloup  
Construction d'un gloup (par copie)  
Construction d'un gloup (par copie)  
Construction d'un gloup  
Destruction d'un gloup  
Construction d'un gloup (par copie)  
Destruction d'un gloup  
Destruction d'un gloup  
Destruction d'un gloup  
Destruction d'un gloup  
Destruction d'un gloup  
[Trotinette:~/tmp] yunes%
```

- et l'effet du passage par référence ?

- Le cas des valeurs de retour...

```
#include "Gloup.hpp"
```

```
Gloup f(Gloup &g1, Gloup &g2)
{
    Gloup *pg = new Gloup;
    delete pg;
    return g1;
}
```

```
int main() {
    Gloup g1, g2;
    g1 = f(g1, g2);
}
```

```
[Trotinette:~/tmp] yunes% ./Gloup
Construction d'un gloup
Construction d'un gloup
Construction d'un gloup
Destruction d'un gloup
Construction d'un gloup (par copie)
Destruction d'un gloup
Destruction d'un gloup
Destruction d'un gloup
[Trotinette:~/tmp] yunes%
```

- et l'effet d'un retour par référence ?

- Le cas des valeurs de retour...

```
#include "Gloup.hpp"
```

```
Gloup &f(Gloup &g1, Gloup &g2)
{
    Gloup *pg = new Gloup;
    delete pg;
    return g1;
}
```

```
int main() {
    Gloup g1, g2;
    g1 = f(g1, g2);
}
```

```
[Trotinette:~/tmp] yunes% ./Gloup
Construction d'un gloup
Construction d'un gloup
Construction d'un gloup
Destruction d'un gloup
Destruction d'un gloup
Destruction d'un gloup
[Trotinette:~/tmp] yunes%
```

- On a tout intérêt à utiliser lorsque c'est possible un passage par référence (pour les objets).
- Note : on peut le *protéger* par un `const`

- Reste un problème...
- Que faire avec les copies explicites, *i.e.* l'affectation ?
- On traitera le problème plus tard... (voir la surcharge d'opérateurs et les opérateurs d'affectation)



LES OBJETS CONSTANTS

- Idée : définir des objets non-mutables, des « constantes » de l'ensemble correspondant
- un point origine pour des points
- la constante π pour les nombres en précision arbitraire...
- la couleur rouge pour les couleurs
- un billet de 100€

- Rappel : les variables **const** sont des variables comme les autres, mais dont l'usage est restreint à un sous-ensemble des actions possibles.
- Le sous-ensemble est défini comme étant l'ensemble des actions qui ne modifient pas la valeur...
- Pour un type ordinaire : grossièrement pas d'affectation, ni incrémentation...
- Pour un type classe ?

- Constantes de type défini par une classe
- L'ensemble des actions accessibles sur un objet considéré comme constant est spécifié en ajoutant à chaque action l'attribut `const`

```
class Nombre {  
public:  
    Nombre(...);  
    long toLong() const;  
    int toInt() const;  
    void set(long v);  
    void set(int v);  
};
```

```
int main() {  
    Nombre n(...);  
    n.set(45L);  
    const Nombre zero(...);  
    zero.toLong();  
    zero.set(23);  
}
```

- Les attributs mutables
 - La contrainte peut être trop forte... On veut pouvoir modifier certains attributs d'un objet alors même qu'il est extérieurement (logiquement) considéré comme constant...
 - Exemple : un compte en banque dont on imagine que les transactions soient toutes tracées et conservées en son sein, y compris les consultations... mais dont la *valeur* ne change pas...

- dans le cas de la consultation (méthode `getPosition`), il est raisonnable de considérer cette méthode comme `const`
- le problème est que certains attributs doivent être modifiés par la méthode `getPosition` (attribut `traces`)
- en ce cas, l'attribut peut être marqué comme `mutable`

```
class Compte {  
private:  
    int position;  
public:  
    int getPosition() const { return position; }  
};
```

```
class Compte {  
private:  
    int position;  
    int nbTransactions;  
public:  
    int getPosition() const {  
        nbTransactions++;  
        return position;  
    }  
};
```

```
class Compte {  
private:  
    int position;  
    mutable int nbTransactions;  
public:  
    int getPosition() const {  
        nbTransactions++;  
        return position;  
    }  
};
```


LES ATTRIBUTS CONSTANTS

- Il est naturel d'avoir à définir des attributs constants
 - exemple : le numéro de sécurité sociale d'un individu
- dans ce cas, il suffit de rajouter à sa définition la qualification `const`
- comment les initialiser puisqu'aucune affectation n'est autorisée ?

```
class Individu{  
private:  
    const int numeroSecuriteSociale;  
};
```

- seule possibilité : une syntaxe d'initialisation particulière (donc à la définition des constructeurs)

```
class Individu{  
private:  
    const int numeroSecuriteSociale;  
public:  
    Individu(...);  
};  
  
Individu::Individu(...) {  
    numeroSecuriteSociale = ...;  
}
```

- seule possibilité : une syntaxe d'initialisation particulière (donc à la définition des constructeurs)

```
class Individu{
private:
    const int numeroSecuriteSociale;
public:
    Individu(...);
};

Individu::Individu(...) : numeroSecuriteSociale(...) {
}
```