

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

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

Moodle (+ groupes):

<https://moodle.u-paris.fr/course/view.php?id=10710>

Attention : vous n'aurez pas d'inscriptions pédagogiques avant 3 semaines (à cause de la liberté du choix des options) Il y a donc risque de surcharge accidentelle dans les groupes. Veuillez déclarer ce que vous faites les premières semaines en choisissant à l'avance un groupe (limité à 30)

Equipe pédagogique :

Aldric Degorre, Yan Jurski, Anne Micheli

aldric.degorre@irif.fr, yan.jurski@irif.fr, anne.micheli@irif.fr

Modalités de contrôle des connaissances :

40% Examen + 40% Projet + 20% TP en session 1

Max (Exam, 40% Exam + 40% Projet + 20% TP) en session 2

Quels prérequis ?

Idéalement vous êtes issus de la L3 d'ici (beaucoup de java, du C, etc ...)

Au moins une bonne connaissance d'un langage de programmation, avec plusieurs projets importants

Quels sont les buts de ce cours ?

- La maîtrise du langage C++ et ses particularités

- (certaines sont contre intuitives, d'autres plus laborieuses)

- Une connaissance de la problématique de l'analyse/conception objet et les solutions apportées par c++ (voir héritage multiple)

- La maîtrise des éléments de base d'UML

- (pas d'algo, de math, ni de preuves)

Éléments abordés :

- Prise en main du langage
- Uml
- Passage d'arguments, copies explicites, affectation
- destruction et petites autres choses (const, static)
- Références croisées, amitié
- Héritage
- Héritage multiple
- Les opérateurs
- La généricité
- Des patrons de conception

INTRODUCTION

Quelques mots sur C++

Une extension du langage C ?

Un langage orienté objet à la syntaxe inspirée du C ?

Quelques caractéristiques :

compatibilité presque totale avec le C (très rares exceptions bien connues, faciles à identifier et à réparer)

performances comparables à celles du C (une obsession de conception) : on « colle » à la machine

Qqs aspects «frustrants/vexants» ... c'est le prix d'avoir fait du neuf avec du vieux : il y a des usages tolérés par le compilateur ou implicite mais déconseillés à présent qu'on oublie facilement.

Dis autrement, on a parfois l'impression de faire de la « psychologie » pour s'adapter à l'outil. (C'est un sentiment, vous verrez cela par vous même)

On se contente de C++ 11 qui a été une évolution importante

L' exemple que vous attendez :

```
#include <iostream>
using namespace std;
int main() {
    cout << "Bonjour tout le monde" << endl;
    return EXIT_SUCCESS;
}
```

Ecrit dans un fichier test.cpp

Compilé dans la console par :

```
g++ --std=c++11 -Wall -o go test.cpp
```

Exécuté par :

```
./go
```


Un exemple original :

```
#include <iostream>
using namespace std;
int main() {
    cout << "Bonjour tout le monde" << endl;
    return EXIT_SUCCESS;
}
```

Ecrit dans un fichier test.cpp

notez l'extension.cpp

Compilé dans la console par :

g++ --std=c++11 -Wall -o go test.cpp

notez les options

Exécuté par :

./go

Un exemple original :

```
#include <iostream>
using namespace std;
int main() {
    cout << "Bonjour tout le monde" << endl;
    return EXIT_SUCCESS;
}
```

EXIT_SUCCESS est la constante 0 (EXIT_FAILURE vaut 1)

cout, endl (et cin pour les entrées clavier) sont des objets constants définis dans la librairie iostream

<< est un opérateur qui associe à gauche

Un exemple original :

```
#include <iostream>
// using namespace std;
int main() {
    std::cout << "Bonjour tout le monde" << std::endl;
    return EXIT_SUCCESS;
}
```

on peut (ou pas) ajouter std au domaine de nom

Si on ne le fait pas, il faut préciser le chemin complet qui permet de retrouver l'objet dont on parle.

Un exemple de lecture clavier :

```
#include <iostream>
int main() {
    using namespace std; // ici seulement sur le bloc
    string name;
    cin >> name;
    cout << "salut à toi " << name << endl;
    return EXIT_SUCCESS;
}
```

notez que cin est utilisé avec >> et que pour cout c'est <<

formellement il faudrait aussi écrire #include <string>
mais là iostream s'en charge (coup de chance)

Le domaine de nom pour string est aussi std (langage de « base »)

Quelques petites choses pour vous perturber (déjà)

```
#include <iostream>
using namespace std;
int main() {
    bool b1=true, b2=false;
    cout << b1 << b2 << endl;    // affiche 10
    cout << boolalpha;
    cout << b1 << b2 << endl;    // affiche truefalse
}
```

(1) le compilateur laisse généreusement passer l'absence de return.

A l'avenir vous serez perturbés par ses optimisations, par les choses qu'il fait par défaut ... parfois vous trouverez cela sympa, parfois vous trouverez ses refus psycho-rigides, d'autres fois vous vous demanderez ce qu'il fabrique :)

(2) std::boolalpha n'est pas affiché, l'opérateur << traite son type comme un modificateur de stream. (Les surcharges peuvent être contre-intuitives.)

Un autre exemple de surcharge :

```
#include <iostream>
using namespace std;
void afficher(int n) {
    cout << "int:" << n << endl;
}
void afficher(float f) {
    cout << "float:" << f << endl;
}
int main() {

    int i=13; float f=3.14159;
    afficher(i);
    afficher(f);
    return EXIT_SUCCESS;
}
```

int:13

float:3.14159

et par ailleurs l'opérateur << est également surchargé ...

Un autre exemple (surcharge.cpp) :

```
#include <iostream>
using namespace std;
/* void afficher(int n) {
    cout << "int:" << n << endl;
}*/
void afficher(float f) {
    cout << "float:" << f << endl;
}
int main() {

    int i=13; float f=3.14159;
    afficher(i);
    afficher(f);
    return EXIT_SUCCESS;
}
```

float:13

float:3.14159

Un autre exemple (surcharge.cpp) :

```
#include <iostream>
using namespace std;
void afficher(int n) {
    cout << "int:" << n << endl;
}
/* void afficher(float f) {
    cout << "float:" << f << endl;
}*/
int main() {

    int i=13; float f=3.14159;
    afficher(i);
    afficher(f);
    return EXIT_SUCCESS;
}
```

int:13

int:3

UNE CLASSE

POUR SE FAIRE UNE IDÉE ...

fichier : exemple3.cpp

```
#include <iostream>
using namespace std;
// définition (et déclaration simultanée) de la classe
class Point {
public:
    int abs, ord;
    Point(int x,int y) : abs{x}, ord{y} {}
    void deplacer(int dx,int dy) { abs+=dx; ord+=dy; }
    void affiche() { cout << '(' << abs << ',' << ord << ')' << endl; }
};
// utilisation
int main() {
    Point p{5,10}; // déclaration + construction
    Point q{5,10};
    p.affiche();   // appels de méthode
    p.deplacer(10,20);
    p.affiche();   // donne (15,30)
    q.affiche();   // donne (5,10)
    return 0;
}
```

Une classe caractérise un ensemble d'éléments qu'on appelle ses objets. (Il y a une classe Point et possiblement de nombreux objets Point. Ici p est un objet de la classe Point)

Cette abstraction permet de regrouper syntaxiquement des choses homogènes, en les écrivant dans un même bloc : `class Point {...}`

On distingue la description et le comportement. Les données qui caractérisent un objet sont appelées ses attributs (ici `abs` et `ord`). Ce qu'on peut demander aux objets est appelé méthodes, ce sont les opérations auxquelles on s'intéresse sur ces objets. Ici nous avons `déplacer` et `affiche`.

On met à part les « constructeurs », qui sont les moyens mis en place pour créer/initialiser les objets. Syntaxiquement ils portent le nom de la classe et reçoivent des arguments. En C++ l'initialisation à la construction se fait via un bloc d'initialisation. Notez la syntaxe avec les « : » et les accolades (nous reviendrons dessus plus tard)

Dans le `main`, on déclare et initialise simultanément l'objet Point `p`

Remarquez la façon dont se passe l'appel d'une méthode. Il faut comprendre que c'est l'objet à gauche du `'.'` qui aura la charge d'exécuter les instructions de la méthode. Ainsi, dans le corps de `"affiche"`, `abs` est l'attribut de l'objet `a` qui on demande d'exécuter `affiche`.

POUR LE PASSAGE À L'ECHELLE IL VA FALLOIR S'ORGANISER:

SÉPARER LES FICHIERS,
RENDRE LISIBLE L'ESSENTIEL,

NE RECOMPILER QUE CE QUI EST NECESSAIRE
(C.A.D. LES NOUVEAUX CHANGEMENTS)

La définition (ou implémentation) consiste en l'écriture détaillée du code, c'est à dire la manière dont les opérations sont effectuées en séquence. Par convention elle sera écrite dans un fichier d'extension `nom.cpp`.

La déclaration consiste en la partie descriptive, elle regroupe : les attributs des objets, les signatures des méthodes et constructeurs, des indications sur la visibilité (`private/public` etc ...) Par convention elle sera écrite dans un fichier d'extension `nom.hpp`.

S'il s'agit de l'implémentation d'une classe, naturellement nous choisirons `nomDeLaClasse.cpp`, et `nomDeLaClasse.hpp`

Enfin, on distingue encore l'application ou la classe sera utilisée, cela permet d'avoir une portabilité claire de la classe, bien isolée dans le couple `hpp/cpp`.

Initialement on avait exemple3.cpp

```
#include <iostream>
using namespace std;
// définition (et déclaration simultanée) de la classe
class Point {
public:
    int abs, ord;
    Point(int x,int y) : abs{x}, ord{y} {}
    void deplacer(int dx,int dy) { abs+=dx; ord+=dy; }
    void affiche() { cout << '(' << abs << ',' << ord << ')' << endl; }
};
// utilisation
int main() {
    Point p{5,10}; // déclaration + construction
    Point q{5,10};
    p.affiche();   // appels de méthode
    p.deplacer(10,20);
    p.affiche();   // donne (15,30)
    q.affiche();   // donne (5,10)
    return 0;
}
```

Partie relative à l'application : un fichier Test.cpp

```
#include "Point.hpp" // on importe la description (elle arrive dans le slide suivant),  
                    // qui est la seule utile au compilateur  
                    // pour vérifier les erreurs de syntaxe  
                    // Remarquez les "" (vos fichiers) et pas <> (la librairie)  
  
// utilisation  
int main() {  
    Point p{5,10}; // déclaration + construction  
    Point q{5,10};  
    p.affiche();   // appels de méthode  
    p.deplacer(10,20);  
    p.affiche();   // donne (15,30)  
    q.affiche();   // donne (5,10)  
    return 0;  
}
```

Partie Déclaration : un fichier Point.hpp

```
class Point {    // pas de code explicite ici
public:
    int abs, ord;
    Point(int x,int y) ;
    void deplacer(int dx,int dy) ;
    void affiche() ;
};
```

(Il manque une petite chose, voir 3 transparents plus loin)

Partie Implémentation : un fichier Point.cpp

```
#include "Point.hpp"    // le code suivant fait référence aux déclarations de ce .hpp
#include <iostream>      // c'est ici qu'on aura besoin de iostream

// Toutes les signatures sont reprises en les préfixant par Point::etc
Point::Point(int x,int y) : abs{x}, ord{y} { }

void Point::deplacer(int dx,int dy) { abs+=dx; ord+=dy; }

void Point::affiche() {
    using namespace std; // pour ne pas exporter ce using, il peut rester local au bloc
    cout << '(' << abs << ',' << ord << ')' << endl;
}
```

Ces 3 fichiers sont maintenant écrits séparément.

On peut demander au compilateur de compiler Test.cpp indépendamment de Point.cpp : il garde des "liens ouverts" vers les méthodes attendues déclarées. Ainsi on peut lancer :

```
g++ -c -Wall -std=c++11 Test.cpp  
(avec l'option -c, produit Test.o)
```

Puis :

```
g++ -c -Wall -std=c++11 Point.cpp  
(avec l'option -c, produit Point.o)
```

"L'édition des liens" (le collage des deux parties) se fait lorsqu'on construit l'exécutable :

```
g++ -o go -Wall -std=c++11 Point.o Test.o  
(sans l'option -c, et ici avec l'option -o pour donner un nom de  
lancement agréable)
```

QQs remarques :

on ne compile pas le hpp ?

non, d'ailleurs il est lui même inclus dans le cpp, ce include est un include hard, fait par le pré-compilateur : include rapatrie le code là où vous avez écrit include.

La séparation hpp/cpp est-elle toujours propre ?

Malheureusement vous pouvez faire ce que vous voulez : écrire vos fichiers avec une autre extension, mettre une partie de code dans la déclaration, définir une classe dans le cpp etc ... cela peut conduire à des pbs : un code défini dans le hpp sera recopié autant de fois qu'il y a un include la seconde redéfinition sera interdite et considérée comme une erreur (et si vous déclarez une classe dans un cpp serait-ce pour faire un include de .cpp ensuite ?)

Un autre pb peut intervenir lors d'inclusions croisées : si A.hpp utilise dans ses signatures B.hpp qui lui aussi utilise A.hpp. Comme l'inclusion est traitée comme une insertion de code, le pré-compilateur va boucler. Pour que l'inclusion soit bien unique, on trouve donc des gardes dans tous les .hpp de la forme :

```
#ifndef TOTO_H
#define TOTO_H
... écrire ici les prototypes ...
#endif
```

Partie Déclaration (avec garde): un fichier Point.hpp

```
#ifndef _POINT // séquence pour assurer l'unicité de la déclaration de cette classe
#define _POINT
class Point { // pas de code explicite ici
public:
    int abs, ord;
    Point(int x,int y) ;
    void déplacer(int dx,int dy) ;
    void affiche() ;
};
#endif
```

(Remarque : le nom de la constante `_POINT` est "librement choisi", mais vous voyez quelles conventions on utilise)

La compilation à la main devient compliquée :

```
g++ -c -Wall -std=c++11 Test.cpp
```

```
g++ -c -Wall -std=c++11 Point.cpp
```

```
g++ -o go -Wall -std=c++11 Point.o Test.o
```

```
./go
```

Nous allons utiliser make couplé à Makefile dont il faut connaître le principe de fonctionnement.

Une fois paramétré vous vous contenterez de lancer :

```
make
```

Le couple make / Makefile :

Makefile est un fichier qui décrit des dépendances générales sous forme d'un arbre dont la syntaxe est

```
noeud : fils1 fils2 ...  
      commande
```

Typiquement dans notre exemple nous avons au moins la dépendance :

```
Point.o : Point.cpp Point.hpp  
g++ -c -Wall -std=c++11 Point.cpp
```

cela signifie qu'un noeud (appelé Point.o) dépend des changements apportés à Point.cpp ou à Point.hpp (Un fils peut être un fichier, comme ici, ou un autre noeud)

S'il y a une nouveauté, il faut rafraîchir la version du noeud (la décision est prise sur le timestamp des fichiers) les fils sont d'abord "résolus" de gauche à droite, puis la commande correspondante à la dépendance est exécutée. Ici le produit sera un fichier Point.o (et on appelle le noeud source de la même façon pour rester simple)

Fichier Makefile

```
all : Test.o Point.o
    g++ -Wall -std=c++11 -o go Test.o Point.o
    ./go

Point.o : Point.cpp Point.hpp
    g++ -c -Wall -std=c++11 Point.cpp

Test.o : Test.cpp Point.hpp
    g++ -c -Wall -std=c++11 Test.cpp

clean :
    rm -f *.o go
```

Remarques :

- syntaxiquement les espaces et les tabulations sont importants !
- notez la dépendance Point.hpp dans les fils de Test.o
- l'absence de dépendance pour clean
- les 2 commandes pour all
- lorsque vous ajoutez une classe il faut modifier le Makefile
- l'objectif est de lancer simplement : make

Simplifications (1 - Factoriser la commande de compilation)

```
CC= g++ -Wall -std=c++11 # compilateur + options
```

```
CCO= $(CC) -c
```

```
all : Test.o Point.o
```

```
    $(CC) -o go Test.o Point.o
```

```
    ./go
```

```
Point.o : Point.cpp Point.hpp
```

```
    $(CCO) Point.cpp
```

```
Test.o : Test.cpp Point.hpp
```

```
    $(CCO) Test.cpp
```

```
clean :
```

```
    rm -f *.o go
```


Simplifications (2 - Améliorer la lisibilité de la compilation finale)

```
CC= g++ -Wall -std=c++11 # compilateur + options
CCO= $(CC) -c
OBJECTS= Test.o Point.o # liste des objets intermédiaires

all : $(OBJECTS)
    $(CC) -o go $(OBJECTS)
    ./go

Point.o : Point.cpp Point.hpp
    $(CCO) Point.cpp

Test.o : Test.cpp Point.hpp
    $(CCO) Test.cpp

clean :
    rm -f *.o go
```

Simplifications (3 - Uniformiser la règle à exécuter)

```
CC= g++ -Wall -std=c++11 # compilateur + options
CCO= $(CC) -c $<
OBJECTS= Test.o Point.o # liste des objets intermédiaires

all : $(OBJECTS)
    $(CC) -o go $(OBJECTS)
    ./go

Point.o : Point.cpp Point.hpp
    $(CCO)

Test.o : Test.cpp Point.hpp
    $(CCO)

clean :
    rm -f *.o go
```

Les variables "automatiques" rendent les choses rapidement moins lisibles.
Pour info \$< indique le premier fils de la règle

La maintenance du fichier Makefile devrait à présent être relativement simple.

En cas de besoin, ou si vous avez commis une erreur, vous pouvez retrouver les dépendances en exécutant :

```
g++ -MM *.cpp
```

qui dans notre exemple produit :

```
Point.o: Point.cpp Point.hpp
```

```
Test.o: Test.cpp Point.hpp
```

Vous obtenez presque les règles nécessaires

IMPORTANT : IL FAUT QUE LE MAKEFILE QUE VOUS FOURNISSEZ SOIT PORTABLE, donc même si vous travaillez sur un IDE nous vous demandons que ce que vous rendez soit compilable simplement avec make !

QQS PETITES CHOSES

POUR POUVOIR COMMENCER RAPIDEMENT

En c++, lorsque vous définissez une fonction à plusieurs arguments :

```
void f (int a, char b, string c) {  
    string res{c+' '};  
    if (a<0) a=-a;  
    for (int i=0;i<a;i++) res+=b;  
    cout << res << endl;  
}
```

destinée à être appelée sous la forme : f(5,'*',"coucou");
qui produit ici : coucou *****

sa déclaration (dans le hpp.) peut contenir des "valeurs par défaut"

```
void f (int a =1, char b='X', string c="toto");
```

cela permet des appels de f sans avoir à tout spécifier.

Ainsi :

f() produira : toto X

f(5) produira : toto XXXXX

f('A') produira ?

En c++, lorsque vous définissez une fonction à plusieurs arguments :

```
void f (int a, char b, string c) {  
    string res{c+' '};  
    if (a<0) a=-a;  
    for (int i=0;i<a;i++) res+=b;  
    cout << res << endl;  
}
```

destinée à être appelée sous la forme : f(5,'*',"coucou");
qui produit ici : coucou *****

sa déclaration (préalable ou dans le hpp) peut contenir des "valeurs par défaut"

```
void f (int a =1, char b='X', string c="toto");
```

cela permet de faire appel à f sans avoir à tout spécifier.

Ainsi :

f() produira : toto X

f(5) produira : toto XXXXX

f('A') produira toto XXXX ...65fois... X (car le code ascii de 'A' est 65)

f("test") ne compilera pas

Les arguments fournis viennent donc remplacer les valeurs par défaut de gauche à droite

En c++ vous pouvez utiliser le mot const pour préciser qu'un paramètre ne peut pas être modifiée.

```
void f( int const x) {  
    x++; // cette instruction produira une erreur de compilation  
}
```

Si vous déclarez une variable constante, alors il vous faudra l'initialiser immédiatement

```
int const deux{2};
```

Nous aurons l'occasion de revenir sur const plus tard.

La classe vector fait partie de la bibliothèque STL (Standard Template Library). Elle modélise les tableaux avec des méthodes de gestion intégrées.

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> v { 7, 5, 16, 8 }; // déclaration initialisation
    v.push_back(25);               // ajout

    for (int n : v) cout << n << ", "; // un parcours possible

    for (size_type i=0;i<v.size();i++) // autre parcours
        cout << v[i] << ", ";

    cout << v.at(5);               // lève une exception
    cout << v[5];                 // affiche qq chose ...
}
```

Remarques :

size_type (qui a aussi pour alias size_t) est "à détails techniques près" comparable à long unsigned int

On reprendra la classe STL un peu plus tard, lorsqu'on sera capable de lire les types en détail

Pause intermédiaire

- Accueil - Modalités - Plan général
- Présentation du langage par l'exemple
- Une classe, sa syntaxe
- La séparation hpp/cpp
(en tout cas
déclaration/implémentation)
- Make/makefile (principes + pratique)
- Petites choses utiles pour le 1er TP
(valeur par défaut, const, vector)

La suite :

- Même chose en plus "formel"
- Initialisation des variables
- Pointeurs, allocation/libération
- Namespaces
- Petit Quizz

L'INITIALISATION DES VARIABLES

En C++ il existe plusieurs syntaxes pour l'initialisation ...

```
int i=4;
```

```
int i;      // implicitement initialisé
```

```
int i(4);   // notation fonctionnelle
```

```
int i{4};   // avec liste d'initialisation
```

```
int i={4};
```

Vous pouvez comprendre que, puisque syntaxiquement ces instructions diffèrent, à la compilation les branches d'analyses peuvent avoir leurs propres règles sémantiques.

Ici tout se passe comme on l'attend, mais avec d'autres types on peut avoir des surprises.

La forme préférée en C++11 est {}

Quelques explications

```
int i; // implicitement initialisé ... mais pas à 0
```

avec la déclaration, le constructeur par défaut est invoqué.

```
int i=4;
```

Pour comprendre la subtilité, on pourrait comparer à

```
int i; // déclaration + construction par défaut  
i=4; // operation d'affectation
```

Mais, en pratique, si le compilateur voit les 2 opérations simultanément, il peut "optimiser" et ne pas construire l'état intermédiaire.

Il faut tout de même comprendre comment il fonctionne, et avoir conscience de la différence en théorie.

Quelques explications

```
int i(4); // notation fonctionnelle
```

Elle pré-suppose l'existence d'un constructeur prévu pour un argument.

Ici aussi, on peut noter une petite chose sur laquelle on pourrait buter.
Imaginons :

```
int i();  
i=4; // il y a une erreur ici ...
```

Qu'on pourrait interpréter comme l'initialisation par défaut de i, puis d'une affectation ... mais ce n'est pas le cas.

Ici la compilation échoue avec :

error: assignment of function 'int i()' in i=4;

....

En effet, l'instruction `int i();` est interprétée comme une déclaration d'une fonction de nom 'i', qui ne prendrait aucun argument (), et retournerait un entier.

Dans ce contexte vous comprenez le message d'erreur.

Quelques explications

```
int i{4}; // avec liste d'initialisation
```

La forme préférée en C++11 est avec {}

Les {} définissent un type liste d'initialisation. Ce type peut être utilisé pour définir un constructeur prenant cet argument. Vu son nom vous comprenez qu'on encourage des constructions avec cette forme.

```
int i={4};
```

accepté, mais pourrait être comparé (s'il n'y avait pas d'optimisation) à la construction d'un objet anonyme du type int à l'aide du constructeur utilisant la liste d'initialisation, puis de l'affectation de i par cette valeur.

En C++ il existe plusieurs syntaxes pour l'initialisation ...

```
int i=4;
```

```
int i;      // implicitement initialisé à 0
```

```
int i(4);   // notation fonctionnelle
```

```
int i{4};   // avec liste d'initialisation
```

```
int i={4};
```

En résumé : l'initialisation en c++ est une question qu'on se pose davantage que dans d'autres langage.

Retenez que :

La forme préférée en C++11 est {}