INF3105 – Introduction au langage C++

Éric Beaudry

Université du Québec à Montréal (UQAM)

2013E

Sommaire

1. Introduction
2. Les fondements du langage C++
3. Fonctions
4. Entrées et sorties
5. Mémoire
6. Classes
7. Const
8. Opérateurs

C++ dans INF3105

- L'objectif principal d'INF3105 $\neq$ apprendre le langage C++.
- C++ est plutôt le langage que nous allons utiliser pour mettre en pratique les concepts fondamentaux de structures de données.
- Les séances en classe ne font pas un tour complet de C++.
- Il faut compléter l'apprentissage de C++ dans les labs et dans ses heures de travail personnel.
- Conseil : prenez une journée complète durant un week-end pour faire un tutoriel en ligne sur C++.

Historique

Origine du C++

- Extension **orienté objet** du langage C.
- « ++ » signifit un incrément par rapport à C.
- Développé par Bjarne Stroustrup au Bell labs d'AT&T dans les années 1980.

Standardisation / Normalisation

- Normalisé par ISO (Organisation mondiale de normalisation) depuis 1998.

Influence

- Le C++ est très utilisé en industrie et en recherche (efficacité).
- Le C++ a influencé d'autres langages comme Java et C#.

Caractéristiques et paradigmes

- Multiplateforme.
- Langage de haut niveau (mais plus bas que Java).
- Compilé en langage machine.
- Impératif.
- Fortement typé.
- Orienté objet.
- Procédural.
- Générique.

Exemple de fichier source C++

bienvenue.cc

```
#include <iostream>

// La fonction main est le point d'entree d'execution
int main(){
    std::cout << "Bienvenue au cours INF3105 en C++ !" << std::endl;
    return 0;
}
```

Fichiers sources

Fichiers d'entête (.h, .hpp)

Les fichiers d'**entête** (*header*), ayant pour extension `.h` ou `.hpp`, contiennent généralement des **déclarations**.

Fichiers sources (.cc, .cpp, .c++)

Les fichiers **sources** ayant pour extension `.cc`, `.cpp` ou `.c++`, contiennent généralement les **définitions** (l'implémentation). Ces fichiers peuvent aussi contenir des déclarations.

Déclaration vs Définition

Déclaration

- La compilation se fait en une seule passe (excluant l'édition des liens).
- Tout doit être déclaré avant d'être utilisé.
- Une déclaration ne fait que déclarer l'existence de quelque chose lié à un identificateur (symbole). Exemples : variables, fonctions, classes, etc.

Définition

- La définition est le code des functions, constructeurs, etc.
- Après la compilation, il y a une passe d'édition des liens (linker).
- Tout symbole utilisé doit être défini à l'édition des liens.

Déclaration vs Définition : Exemple 1

helloworld.cc

```
#include <iostream>
int main(int argc, char** argv)
{
    allo(); // Error: symbol allo undefined!
    return 0;
}
// Declaration et definition d'une fonction allo()
void allo(){
    std::cout << "Hello World!" << std::endl;
}
```

Déclaration vs Définition : Exemple 2

helloworld.cc

```
#include <iostream>
// Declaration et definition d'une fonction allo()
void allo(){
    std::cout << "Hello World!" << std::endl;
}
int main(int argc, char** argv)
{
    allo();
    return 0;
}
```

Déclaration vs Définition : Exemple 3

helloworld.cc

```
#include <iostream>
// Declaration de la fonction allo()
void allo();
int main(int argc, char** argv)
{
    allo();
    return 0;
}
// Definition de la fonction allo()
void allo(){
    std::cout << "Hello World!" << std::endl;
}
```

Exemple de fichiers sources

allo.h

```
void allo();
```

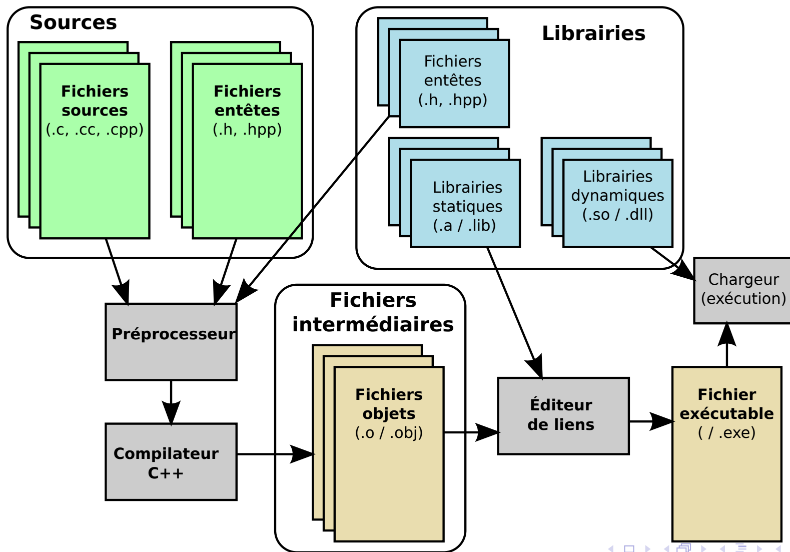
allo.cc

```
#include <iostream>
void allo(){
    std::cout << "Hello World!" << std::endl;
}
```

helloworld.cc

```
#include "allo.h"
void main(int argc, char** argv) {
    allo();
}
```

Organisation et compilation



Quelques mots réservés

Types

void, bool, char, short int, int, long, float, double
unsigned

Boucles et instructions de contrôles

if, else, while, do, for, switch ... case

Structures

class, struct, union

Types

Type (mot clé)	Description	Taille (octets)	Capacité
`bool`	Booléen	1	$\{false, true\}$
`char`	Entier / caractère ASCII	1	$\{-128, \ldots, 127\}$
`unsigned char`	Entier / caractère ASCII	1	$\{0, \ldots, 255\}$
`unsigned short` `unsigned short int`	Entier naturel	2	$\{0, \ldots, 2^16 - 1\}$
`short` `short int`	Entier	2	$\{-2^{15}, \ldots, 2^{15} - 1\}$
`unsigned int`	Entier naturel	4	$\{0, \ldots, 2^{32} - 1\}$
`int`	Entier	4	$\{-2^{31}, \ldots, 2^{31} - 1\}$
`unsigned long`	Entier naturel	8	$\{0, \ldots, 2^{64} - 1\}$
`long`	Entier	8	$\{-2^{63}, \ldots, 2^{63} - 1\}$
`float`	Nombre réel	4	$\pm 3.4 \times 10^{\pm 38}$ (7 chiffres)
`double`	Nombre réel	8	$\pm 1.7 \times 10^{\pm 308}$ (15 chiffres)
`long double`	Nombre réel	8	$\pm 1.7 \times 10^{\pm 308}$ (15 chiffres)

Déclaration variables

Une variable est une **instance** d'un type de données. En C++, les variables sont considérées comme des **objets**. Chaque variable est nommée à l'aide d'un identificateur. L'identificateur doit être unique dans sa portée.

Exemple

```
// Declaration d'un entier (sans initialisation)
int a;
/* Declaration de nombres reels (sans initialisation) */
float f1, f2, f3;
/* Declaration de nombres avec initialisation explicite*/
int n1(12), n2=20;
```

Initialisation des variables

- Constructeur : lors d'une initialisation explicite.
- Constructeur sans argument : si aucune initialisation n'est explicitée.
- Par défaut, les types de base ne sont pas initialisés.
    - Avantage : Efficacité.
    - Inconvénient : L'exécution peut dépendre du contenu précédent en mémoire.
    - Inconvénient : L'exécution peut être pseudo non déterministe (comportement « aléatoire »).
    - Problème : Source potentielle de bugs.

Énoncés et expressions

Comme dans la plupart des langages de programmation, le corps d'une fonction en C++ est constitué d'énoncés (*statements*). Sommairement, un énoncé peut être :

- une déclaration de variable(s) ;
- une expression d'affectation ;
- une expression ;
- une instruction de contrôle ;
- un bloc d'énoncés entre accolades `{ }`.

À l'exception d'un bloc `{ }`, un énoncé se termine toujours par un point-virgule (`;`).

Énoncés / Affectation

Affectation

```
// Declaration
int a;
// Affectation
a = 2 + 10;
```

Expressions

En C++, une expression peut être :

- un identificateur (variable) ou un nombre ;
- une expression arithmétique ou logique ;
- un appel de fonction ;
- une autre expression entre parenthèses ( ) ;
- un opérateur d'affectation (=, +=, etc.) ;
- etc.

Exemples d'expressions

```
4+5*6-8;
(4+5)*(6-8);
a * 2 + 10;
```

```
a = b = c = d;
// est l'equivalent de :
c = d; b = c; a = b;
```

Exemples d'expressions

```
a++; // a = a + 1;
a+=10; // a = a + 10;
a*=2; // a = a * 2;
a/=2; // a = a / 2;
```

```
b = a++; // b=a; a=a+1;  // post-increment
b = ++a; // a=a+1; b=a;  // pre-increment
b = a--; // b=a; a=a-1;  // post-decrement
b = --a; // a=a-1; b=a;  // pre-increment
```

Instructions de contrôle

if, while, for, do...while, switch ... case, break, ...

Tableaux

```
int tableau1[5] = {0, 5, 10, 15, 20};
int tableau2[10] = {0, 5, 10, 15, 20};
int tableau3[] = {0, 5, 10, 15, 20};
```

Non-vérification des indices des tableaux

Essayez :

```
#include <iostream>
using namespace std;
int main() {
  int tab1[5], tab2[5];
  for(int i=0;i<5;i++){
    tab1[i] = i;    tab2[i] = i + 10;
  }
  for(int i=0;i<16;i++)  cout << " " << tab1[i];
  cout << endl;
  for(int i=0;i<15;i++)  tab1[i] = 99 - i;
  for(int i=0;i<5;i++)   cout << " " << tab1[i];
  cout << endl;
  for(int i=0;i<5;i++) cout << " " << tab2[i];
  cout << endl;
  return 0;
}
```

Non-vérification des indices des tableaux

- Aller chercher un indice dans tableau se fait par une arithmétique de pointeurs.
- Exemple : `tab2[10]` est équivalent à `*(tab2 + 10)`.
- Note : le `+10` est implicitement multiplié par `sizeof(int)` à la compilation.
- La non-vérification des indices = Source potentielle de bugs.

Pointeurs et références

- Pointeur = adresse mémoire.
- Pointeurs différents en Java.
- Référence = encapsulation d'un pointeur utilisable comme un objet.
- Passage de paramètres par valeur ou par référence.
- Le passage par pointeur est un passage par valeur d'une adresse pointant vers un objet donné.

Pointeurs

Dans la déclaration de variable, la porté d'un symbole étoile * se limite à une variable.

```
int n = 3;
int* ptr_n = &n;
int* tableau = new int[100];
```

```
//Declare le pointeur p1 et l'objet o1
int* p1, o1;
//Declare les pointeurs p2, p3, p4 et l'objet o2
int *p2, *p3, o2, *p4;
```

Déférencement de pointeurs

Déférencer = aller chercher (le contenu de) la case mémoire.

```
int n=0;
int *pointeur = &n;
*pointeur = 5; // effet : n=5
std::cout << "n=" << *pointeur << std::endl;
```

Arithmétique des pointeurs

Code 1 (lisibilité)

```
int tableau[1000];
int somme = 0;
for(int i=0;i<1000;i++)
    somme += tableau[i];
```

Code 2 (efficacité*)

```
int tableau[1000];
int somme = 0;
int* fin = tableau+1000; // pointe sur l'element suivant le dernier element
for(int* i=tableau;i<fin;i++)
    somme += *i;
```

Références

```
int n = 2;
int& ref_n = n;
n = 3;
std::cout << "ref_n=" << ref_n << std::endl;
```

Fonctions

Similaire à Java et C.

Passage de paramètres

```
void test(int a, int* b, int* c, int& d, int*& e){
    a=11; // effet local
    b++; // change l'adresse locale de b
    *c=13; // change la valeur pointee par c
    d=14; //  change la valeur referee par d
    e=c; // change la valeur du pointeur (adresse) pour celle de c.
}
int main(){
    int v1=1, v2=2, v3=3, v4=4, *p5=&v1;
    test(v1, &v2, &v3, v4, p5);
    cout<<v1<<'\t'<<v2<<'\t'<<v3<<'\t'<<v4<<'\t'<<*p5<<'\t'<<endl;
    // affiche : 1  2  13  14 13
    return 0;
}
```

Entrée standard et sortie standard

Pour avoir accès aux flux standards de la bibliothèque standard de C++, il faut inclure le fichier d'entête `iostream`. Ce dernier définit les trois flux suivant :

- `std::cin` : flux d'entrée depuis l'entrée standard (*stdin*) ;
- `std::cout` : flux de sortie vers la sortie standard (*stdout*) ;
- `std::cerr` : flux de sortie vers la sortie d'erreurs (*stderr*).

Exemple

demo01.cc

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

void main(int argc, char** argv){
  int a, b;
  cout << "Entrez deux nombres:" << endl;
  cin >> a >> b;
  int somme = a + b;
  cout << "La somme est " << somme << endl;
}
```

demo02.cc

```
#include <fstream>
void main(int argc, char** argv){
  int a, b;
  std::ifstream in("nombres.txt");
  cout << "Lire deux nombres:" << endl;
  in >> a >> b;
  int somme = a + b;
  std::ostream out("somme.txt");
  out << somme << endl;
}
```

Tas (*heap*)

Pile d'exécution

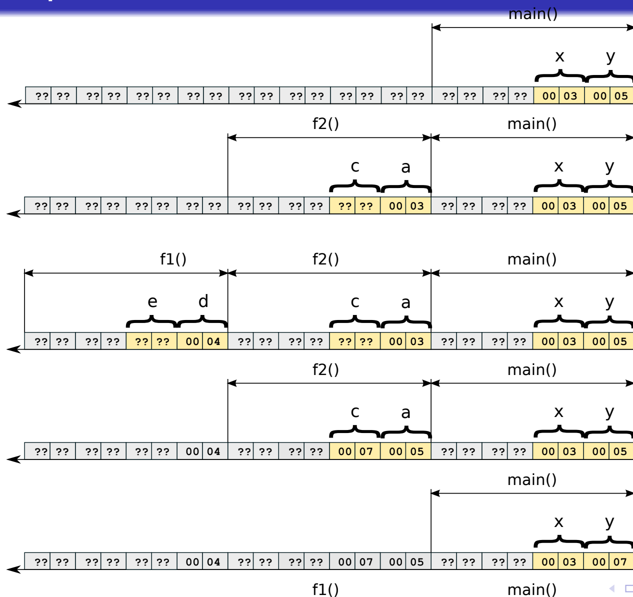
Allocation mémoire

- Automatique (fait automatiquement par le compilateur).
- Dynamique (allouée explicitement sur le tas (heap)).
- Toute mémoire allouée dynamiquement doit être libérée dynamiquement.
- Algorithmes d'allocation/libération (dépendant du compilateur/système d'exploitation).

Allocation sur la pile

```
short int f1(){
   int d = 4;   int e;
   return d;
}
short int f2(short int a){
   int c = a + f1();
   a += 2;
   return c;
}
void main(){
   short int x=3;   short int y=5;
   y = f2(x);
   f1();
}
```

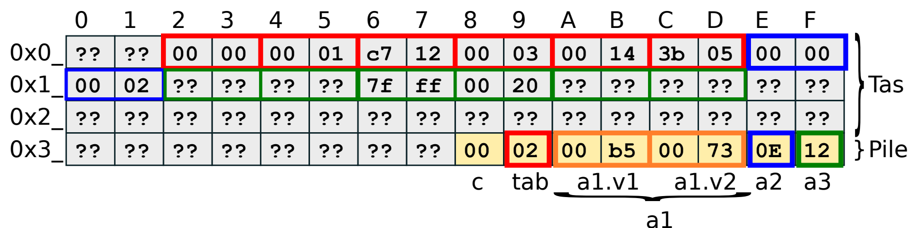
Espace mémoire



Allocation sur le tas (*heap*)

```
struct A{
    short int v1, v2;
};
void main(){
    char c = 0;
    short int *tab = new short int[6] {0x00, 0x01, 0xc712, 0x03, 0x14, 0x3b05};
    A a1; a1.v1=0x00b5; a1.v2=0x0073;
    A* a2 = new A();
    a2->v1=0; a2->v2=2;
    A* a3 = new A[3];
    a3[1].v1=0x7fff; a3[1].v2=0x0020;
    // Sur la diapo suivante : etat de la memoire jusqu'ici
    delete[] tab;    delete a2;    delete[] a3;
}
```

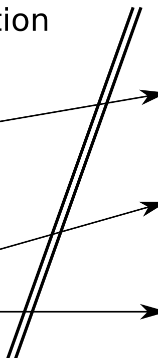
Allocation mémoire



Représentation abstraite de la mémoire

Pile d'exécution

c		0
tab		
a1	.v1	00b5
	.v2	0173
a2		
a3		



Tas (*Heap*)

00	01	c712	03	14	3b05

0
2

?	7fff	?
?	20	?

Classes en C++

- Similaire à Java
- En fait Java est inspiré (et simplifié !) de C++.
- Constructeurs.
- Constructeur par défaut.
- Constructeur par copie.
- Destructeurs.
- Surcharge d'opérateurs (+, -, +=, -=, =, ==, <, (), etc.).

Classe Point

```
class Point {
  public:
    double distance(const Point& p) const;
  private:
    double x, y;
};
```

Constructeurs

Un constructeur porte le nom de la classe et peut avoir zéro, un ou plusieurs arguments. Comme son nom l'indique, le rôle d'un constructeur est de construire (instancier) un objet. Un constructeur effectue dans l'ordre :

1. appelle le constructeur de la ou des classes héritées ;
2. appelle le constructeur de chaque variable d'instance ;
3. exécute le code dans le corps du constructeur.

Destructeurs

Un constructeur porte le nom de la classe et peut avoir zéro, un ou plusieurs arguments. Comme son nom l'indique, le rôle d'un constructeur est de construire (instancier) un objet. Un constructeur effectue dans l'ordre :

1. appelle le constructeur de la ou des classes héritées ;
2. appelle le constructeur de chaque variable d'instance ;
3. exécute le code dans le corps du constructeur.

Déclaration

```
class Point {
 public:
   Point(); // constructeur sans argument
   Point(double x, double y);
   ...
};
```

Définition

```
Point::Point(){
   x = y = 0.0;
}
Point::Point(double x_, double y_)
 : x(x_), y(y_) // le deux−points (":") est pour l'initialisation
{
}
```

Classe avec gestion de mémoire

```
class Tableau10int{
  public:
    Tableau10int();
    ~Tableau10int();
  private:
    int* elements;
};
```

Constructeur et Destructeur

```
Tableau10int::Tableau10int() {
    elements = new int[10];
}
Tableau10int::~Tableau10int() {
    delete [] elements ;
}
```

Héritage et Fonctions virtuelles

```
class FormeGeometrique{
 public:
  virtual double aire() = 0;
};
class Carre : public FormeGeometrique{
 public:
  Carre(double dimension) { m_dimension(dimension); }
  virtual double aire() {return m_dimension*m_dimension; }
 protected:
  int m_dimension;
};
class Rectange : public FormeGeometrique{
 public:
  Rectange(double h, double l) { m_hauteur(h); m_largeur(l); }
  virtual double aire() {return m_hauteur*m_largeur; }
 protected:
  int m_hauteur, m_largeur;
};
```

Mot clé `this`

- Le pointeur `this` pointe sur l'objet courant
- C'est un paramètre implicite.

```
class A{
  public:
    int f(int v=0);
  private:
    int a;
};
int A::f(int v){
    int t = this->a; // t=a
    this->a = v; // a=v
    return a;
}
```

Mot clé `const`

- Le mot clé `const` est très important en C++.
- L'objectif est d'aider le programmeur à éviter des bogues.
- Permet de spécifier que quelque chose ne doit pas être modifié.
- Si le programme tente de modifier un objet `const`, le compilateur va générer une erreur.
- Important : `const` ne doit être utilisé comme un dispositif de sécurité. Le mécanisme de `const` est seulement une aide au programmeur.

Contexte

- Utile pour spécifier :
  - une constante (ex. : `const double pi=3.141592654;`)
  - que l'objet référencé par une référence doit être constant (ex. : `const Point& rp = p;`)
  - que l'objet pointé par un pointeur ne doit pas être modifié (ex. : `const Point* rp = &p;`)
- Très utilisé dans le passage de paramètres.

Exemple

```
double Point::distance(Point& p2){
    p2.x -= x; y-=p2.y;
    return sqrt(p2.x*p2.x + y*y);
}
int main(){
    Point p1 = ... , p2 = ...;
    double d = p1.distance(p2);
    // p1 et p2 ont ete modifies par Point::distance(...).
}
```

Exemple

```
double Point::distance(const Point& p2) const{
    p2.x -= x; // genere une erreur car p2 est const
    y-=p2.y;   // genere une erreur car *this est const
    return sqrt(p2.x*p2.x + y*y);
}
int main(){
    Point p1 = ... , p2 = ...;
    double d = p1.distance(p2);
    // p1 et p2 ont ete modifies par Point::distance(...).
}
```

Exemple

```
double Point::distance(const Point& p2) const{
    double dx = p2.x−x; // OK
    double dy−=p2.y−y;  // OK
    return sqrt(dx∗dx+dy∗dy);
}
int main(){
    Point p1 = ... , p2 = ...;
    double d = p1.distance(p2);
    // p1 et p2 ont ete modifies par Point::distance(...).
}
```

Opérateurs

- Symboles : `+`, `-`, `*`, `/`, `!`, `+=`, `-=`, `^`, `<<`, `>>`, `()`, etc.
- Les opérateurs peuvent être appelés de façon «naturelle».
  - Exemple : au lieu d'écrire `int a = plus(b,c);`, on écrit tout simplement `int a=b+c;`.
  - Exemple : au lieu d'écrire `int a = divise(plus(b,c),2);`, on écrit tout simplement `int a=(b+c)/2;`.
- Nous pouvons faire la même chose avec nos propres types de données.
  - `Vecteur a(10,5), b(5,10); Vecteur c=a+b;` au lieu de `c=plus(a,b);`.

Surcharge d'opérateurs

- Les opérateurs sont des fonctions avec le mot clé `operator` comme préfixe.
- Différence : appel plus naturel qu'un appel de fonction.

vecteur.h

```
class Vecteur {
 public:
   Vecteur(double vx_=0, double vy_=0):vx(vx_),vy(vy_){}
   Vecteur& operator += (const Vecteur& v);
   Vecteur operator + (const Vecteur& v) const;
 private:
   double vx, vy;
};
```

Surcharge d'opérateurs

vecteur.cpp

```
#include "vecteur.h"
Vecteur& Vecteur::operator += (const Vecteur& autre) {
    vx−=autre.vx;  vy−=autre.vy;
    return ∗this;
}
Vecteur Vecteur::operator + (const Vecteur& autre) const{
    return Vecteur(vx+autre.vx, vy+autre.vy);
}
```

Opérateurs << et >> pour les E/S

point.h

```
class Point{
  private:
    double x, y;
    friend std::istream& operator >> (std::istream& is, Point& p);
    friend std::ostream& operator << (std::ostream& os, const Point& p);
};
```

Opérateurs << et >> pour les E/S

point.cpp

```
std::istream& operator >> (std::istream& os, Point& p){
    char parouvr, vir, parferm;
    is >> parouvr >> p.x >> vir >> p.y >> parferm;
    assert(parouvr=='(' && vir==',' && parferm==')');
    return is;
}
std::ostream& operator << (std::ostream& os, const Point& p){
    os << "(" << p.x << "," << p.y << ")";
    return os;
}
```

Exercice d'abstraction

- Un type d'objet doit savoir comment se lire et s'écrire...
- Mais doit faire abstraction des types des objets qui le composent.

```
class Immeuble {
 public:
 private:
   string nom;
   Point  position;
   double hauteur;
   int    nbclients;

   friend std::istream& operator >> (std::istream& is, Immeuble& im);
};
```

Mauvaise approche...

```
std::istream& operator >> (std::istream& is, Immeuble& im){
  is >> im.nom;
  // Debut mauvais code
  char parouvr, vir, parferm;
  is >> parouvr >> im.position.x >> vir >> im.position.y >> parferm;
  assert(parouvr=='(' && vir==',' && parferm==')');
  // Fin mauvais code
  is >> im.hauteur;
  is >> im.nbclients;
  return is;
};
```

Bonne approche...

```
std::istream& operator >> (std::istream& is, Immeuble& im){
  is >> im.nom;
  is >> im.position;
  is >> im.hauteur;
  is >> im.nbclients;
  return is;
};
```

Chaîne d'appels à >>

Version en plusieurs énoncés

```
istream& operator>>(istream&
      is, Immeuble& im)
{
   is >> im.nom;
   is >> im.position;
   is >> im.hauteur;
   is >> im.nbclients;
   return is;
};
```

Version en un seul énoncé

```
istream& operator>>(istream&
      is, Immeuble& im)
{
   is >> im.nom
      >> im.position
      >> im.hauteur
      >> im.nbclients;
   return is;
};
```

Chaîne d'appels à >>

Pourquoi `return is;` ?

Version en un seul énoncé

```
istream& operator>>(istream& is,
      Immeuble& im){
  is >> im.nom
     >> im.position
     >> im.hauteur
     >> im.nbclients;
  return is;
};
```

Équivalence d'appels

```
istream& operator>>(istream& is,
      Immeuble& im){
  operator>>(
    operator>>(
      operator>>(
        operator>>(is, im.nbclients),
        im.hauteur),
      im.position),
    im.nom);
  return is;
};
```