Introduction aux nouveautés du C++ 11

Miss Réunion

Laurent NOËL
laurent.noel.c2ba@gmail.com
http://igm.univ-mlv.fr/~lnoel

07/05/2013

nullptr

Quel est le problème de ce code ?

Code

```
#include <cstdlib>

void f(int x) {
}

void f(void* ptr) {
}

int main() {
    f(NULL);
    return 0;
}
```

nullptr

Erreur

```
In function ‘int main()’:
12:11: error: call of overloaded ‘f(NULL)’ is ambiguous
12:11: note: candidates are:
3:6: note: void f(int)
7:6: note: void f(void*)
```

nullptr

Problèmes de la constante **NULL**:

- De type **int**
- Nécessite le header **cstdlib**

nullptr

nullptr est une constante de type **nullptr_t** destinée à remplacer l'utilisation du typedef **NULL**.

New school

```
#include <cstdlib>

void f(int x) {
}

void f(void* ptr) {
}

int main() {
    f(nullptr);
    return 0;
}
```

auto

Old school

```
std::vector<int> v;
for(std::vector<int>::iterator it = v.begin();
    it != v.end(); ++it) {
    // do something
}
```

New school

```
std::vector<int> v;
for(auto it = v.begin(); it != v.end(); ++it) {
    // do something
}
```

Une variable déclarée **auto** est typée ! Ce n'est pas du typage dynamique.

auto

Très utile lorsqu'on ne connait pas le type de retour d'une fonction:

```
int plus(int a, int b) {
    return a + b;
}

auto plus_5 = std::bind(plus, 5, std::placeholders::_1);

// Affiche 11
std::cerr << plus_5(6) << std::endl;
```

auto

Pointeurs et références:

```
int var = 58;

auto& ref = var;
ref = 12; // var vaut 12

auto* ptr = &var;
*ptr = 0; // var vaut 0
```

std::begin() et std::end()

Old school

```
std::vector<int> v;
for(auto it = v.begin(); it != v.end(); ++it) {
    // do something
}
```

New school

```
std::vector<int> v;
for(auto it = begin(v); it != end(v); ++it) {
    // do something
}
```

std::begin() et std::end()

Fonctionne également avec les tableaux statiques:

New school

```
int t[] = { 1, 2, 3, 4 };
for(auto it = begin(t); it != end(t); ++it) {
    // do something
}
```

L'utilisation de **begin** et **end** en fonctions non-membre permet de rendre le code plus homogène et plus générique. La classe conteneur utilisée n'a plus à fournir les méthodes **begin** et **end** pour être parcourue par itérateur: il suffit de surcharger les version non-membre des fonctions.

Boucle de type for-each

Old school

```
std::vector<int> v;
for(auto it = v.begin(); it != v.end(); ++it) {
    // do something
}
```

New school

```
std::vector<int> v;
for(auto value: v) {
    // do something
}
```

Sucre syntaxique powa :D

Smart pointers

Quel est le problème de ce code ?

Old school

```
void f() {
    int* ptr = new int(0);

    // do something

    delete ptr;
}
```

Smart pointers

Pour éviter les fuites !

New school

```
void f() {
    std::unique_ptr<int> ptr(new int(0));

    // do something
}
```

Le destructeur de **unique_ptr** se charge d'appeler **delete**: c'est l'idiome RAII.
Un object de type **unique_ptr** se comporte comme un pointeur classique: les opérateur `*` et `->` sont définis. Il est possible de récupérer le pointeur brut avec la méthode **get()**.

unique_ptr

Ne peux pas etre copié dans un autre **unique_ptr**

Bad code

```
void f(const std::unique_ptr<int> ptr) {
// do something
}
void g(const std::unique_ptr<int>& ptr) {
// do something
}

std::unique_ptr<int> ptr(new int(0));
f(ptr); // ERREUR, tentative de passage par copie
g(ptr); // OK, passage par reference
```

Permet de s'assurer qu'un pointeur n'est possédé que par un seul **unique_ptr**.

shared_ptr

Contient un compteur de références: lorsque le compteur arrive à 0 l'objet contenu est détruit:

C++11

```
void f(std::shared_ptr<int> ptr) {
// do something
}

std::shared_ptr<int> ptr(new int(0)); // Compteur à 1
f(ptr); // Le compteur passe à 2 dans la fonction

// Sortie de la fonction le compteur repasse à 1
```

Permet de partager la ressource: aucun objet n'est réellement désigné comme le propriétaire.

Bonnes pratiques

Ne pas créer de **shared_ptr** en appelant **new** directement: utiliser **make_shared**:

C++11

```
// Pointeur de type int, valeur initialisée à 0.
auto ptr = std::make_shared<int>(0);

struct Vec3 {
    float x, y, z;
    Vec3(float x, float y, float z): x(x), y(y), z(z) {}
};

// Pointeur de type Vec3, value initialisée à (1, 2, 3)
auto ptr2 = std::make_shared<Vec3>(1, 2, 3);
```

En **C++14**, **make_unique** fera son apparition.

Avec de l'héritage

Les smart pointeurs permettent également le sous-typage:

C++11

```
struct Base {
    virtual ~Base() {
    }
};

struct Derived: public Base {
};

std::shared_ptr<Base> ptr = std::make_shared<Derived>();
```

Move semantic

Quel est le problème de ce code:

C++

```
std::vector<int> computeSequence(int n) {
   std::vector<int> v;
   for(int i = 0; i < n; ++i) {
      v.push_back(i);
   }
   return v;
}

std::vector<int> v = computeSequence(1000000000);
```

Move semantic

Quel est le problème de ce code:

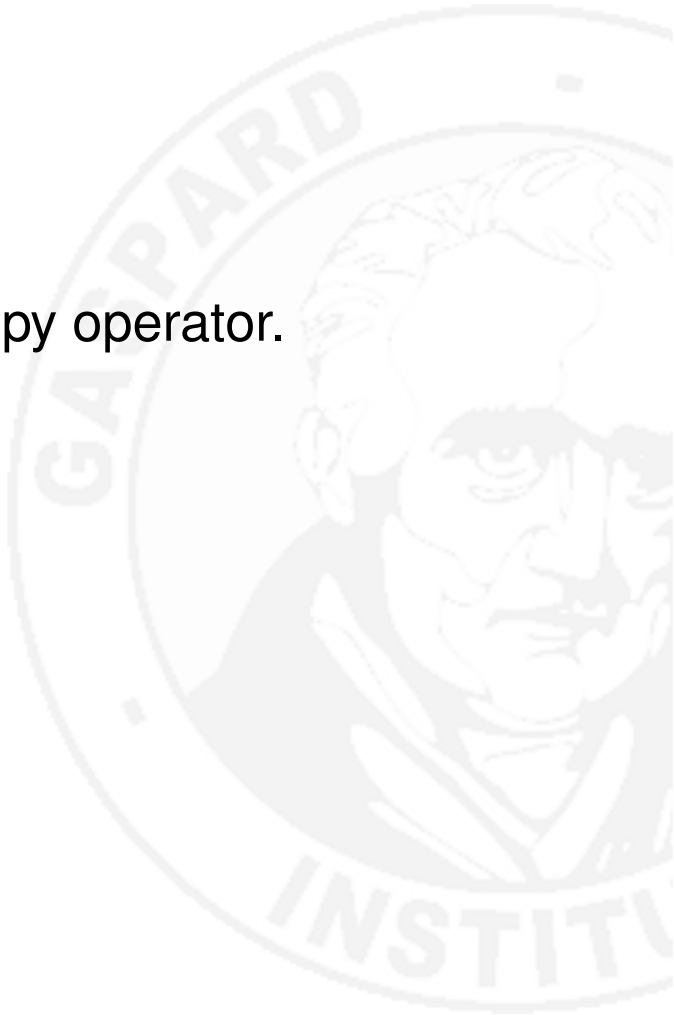
C++

```
std::unique_ptr<int> alloc_int(int value) {
    return std::unique_ptr<int>(value);
}

std::unique_ptr<int> ptr = alloc_int(10);
```

Move semantic

Nouveaux concept: Move constructor et Move copy operator.



Move semantic

```
class BigVector {
    int* m_pData;
public:
    BigVector():
        m_pData(new int[1000000]) {
    }
    ~BigVector() {
        delete [] m_pData;
    }
};
```

Move semantic

```
// Copy constructor: slow
BigVector(const BigVector& v):
    m_pData(new int[1000000]) {
    std::copy(v.m_pData, v.m_pData + 1000000, m_pData);
}
// Copy operator: slow
BigVector& operator =(const BigVector& v) {
    std::copy(v.m_pData, v.m_pData + 1000000, m_pData);
    return *this;
}
```

Move semantic

```
BigVector computeBigVector() {
   BigVector v;

   // do something

   return v;
}

BigVector v = computeBigVector(); // Arg ! Copy
```

Move semantic

```
// Move constructor: fast
BigVector(BigVector&& v):
    m_pData(v.m_pData) {
    v.m_pData = nullptr;
}

// Move operator: fast
BigVector& operator =(const BigVector& v) {
    m_pData = v.m_pData;
    v.m_pData = nullptr;
    return *this;
}
```

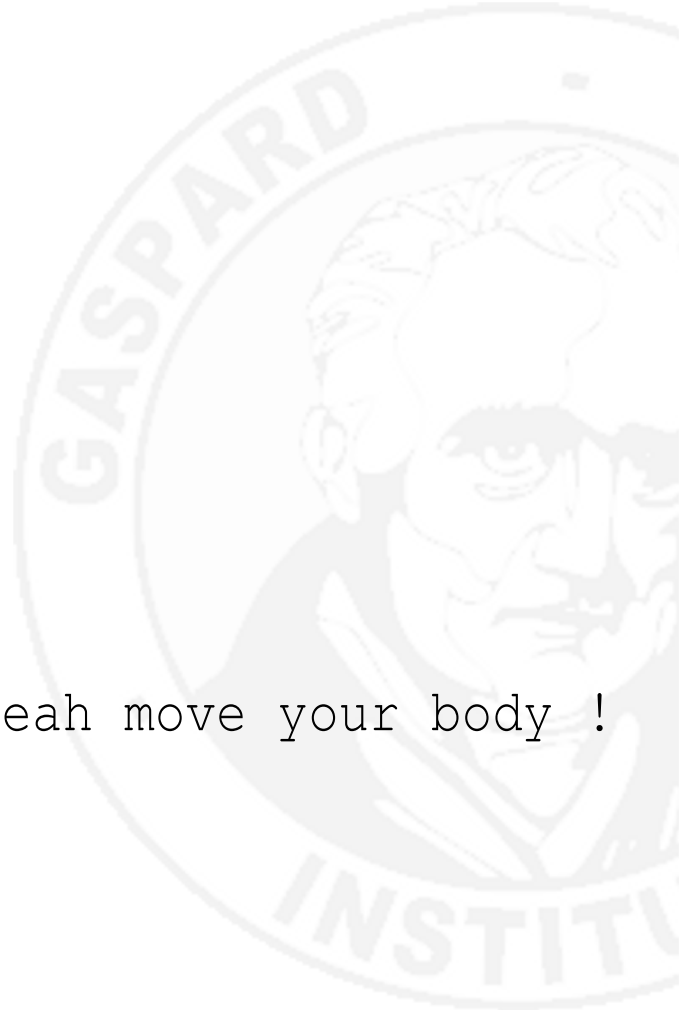
Move semantic

```
BigVector computeBigVector() {
    BigVector v;

    // do something

    return v;
}

BigVector v = computeBigVector(); // Yeah move your body !
```



Move semantic

Forcer le move:

```
BigVector v = computeBigVector();

BigVector copy = v;

BigVector stealer = std::move(v); // Han !

// Attention: ne plus utiliser v ici !
```

Perfect forwarding

Problème:

```
class C {
public:
    void setV(const BigVector& v) {
        m_V = v; // Copy
    }
private:
    BigVector m_V;
};

C c;
c.setV(computeBigVector());
```

Ici on copie alors qu'on pourrait move !

Perfect forwarding

Solution 1:

```
class C {
public:
    void setV(const BigVector& v) {
        m_V = v; // Copy
    }
    void setV(BigVector&& v) {
        m_V = std::move(v);
    }
private:
    BigVector m_V;
};
```

Perfect forwarding

Solution 2: Perfect forwarding

```
class C {
public:
    template<typename T>
    void setV(T&& v) {
        m_V = std::forward(v);
   }
private:
   BigVector m_V;
};
```

Lambdas

```
int RandomNumber () { return std::rand() % 100; }

void f() {
    std::vector<int> v(16);
    std::generate(begin(v), end(v), RandomNumber);
}
```

C'est chiant de définir une fonction pour faire ça, non ? La fonctionnalité requise est très locale.

Lambdas

```
void f() {
    std::vector<int> v(16);
    std::generate(begin(v), end(v),
        []() -> int {
            return std::rand() % 100;
        });
}
```

Ici on utilise un lambda, directement au moment de l'appel à **generate**.

Lambdas

Syntaxe:

```
[capture](args) -> return_type {
    // body
}
```

La capture permet de récupérer des variables locales environnantes:

```
int i = 5;
auto ret_i = [i]() -> int {
    return i;
}
cout << ret_i() << endl; // affiche 5
i = 12;
cout << ret_i() << endl; // affiche 5 aussi !
```

Lambdas

Capture par référence:

```
int i = 5;
auto ret_i = [&i]() -> int {
    return i;
}
cout << ret_i() << endl; // affiche 5
i = 12;
cout << ret_i() << endl; // affiche 12
```

Lambdas

Capturez les tous !

```
int i;
// Capture toutes les variables accessibles par valeur
auto ret_i = [=]() -> int {
    return i;
}
// Capture toutes les variables accessibles par référence
auto set_i = [&](int a) -> int {
    i = a;
}
```

Nouvelle syntaxe pour les types de retour

Exemple:

```
int f(float x) {
    return int(x);
}

// Peut aussi s'écrire:
auto f(float x) -> int {
    return int(x);
}
```

Quel est l'interêt ?

Nouvelle syntaxe pour les types de retour

```
template<typename T>
struct Vec2 {
    T x, y;
    Vec2(T x, T y): x(x), y(y) {}
};
template<typename T>
Vec2<T> operator +(Vec2<T> v1, Vec2<T> v2) {
    return Vec2<T>(v1.x + v2.x, v1.y + v2.y);
}
Vec2<int> vint;
Vec2<float> vfloat;

Vec2<float> r = vint + vfloat; // Ne compile pas
```

Nouvelle syntaxe pour les types de retour

Le C++11 propose l'instruction **decltype(expr)** qui renvoit le type d'une expression.
Ce qu'on voudrait faire:

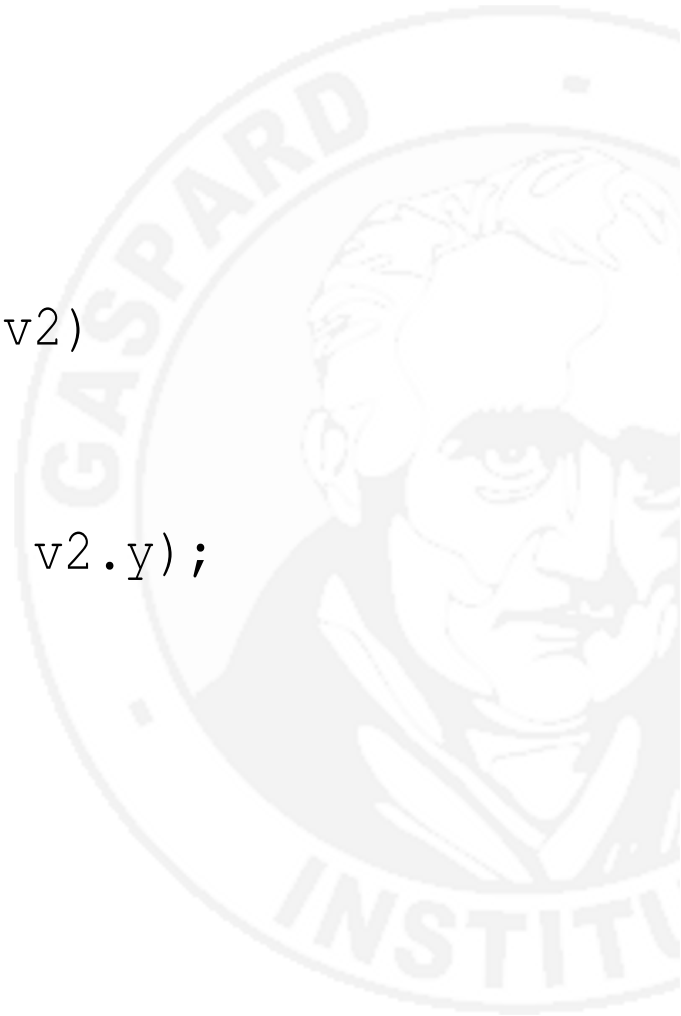
```
template<typename T1, typename T2>
Vec2<decltype(v1.x + v2.x)>
    operator +(Vec2<T1> v1, Vec2<T2> v2) {
    typedef decltype(v1.x + v2.x) T;
    return Vec2<T>(v1.x + v2.x, v1.y + v2.y);
}
```

Mais v1 et v2 ne sont pas déclarées au moment ou le premier **decltype()** est utilisé;

Nouvelle syntaxe pour les types de retour

Il faut donc utiliser la nouvelle syntaxe:

```
template<typename T1, typename T2>
auto operator +(Vec2<T1> v1, Vec2<T2> v2)
    -> Vec2<decltype(v1.x + v2.x)> {
    typedef decltype(v1.x + v2.x) T;
    return Vec2<T>(v1.x + v2.x, v1.y + v2.y);
}
```



Liste non exhaustive des points non traités

- **static_assert** et le header **type_traits**
- les mots clef **override** et **final**
- les types énumérés fortement typés
- les extensions de la STL: **thread**, **atomic**, **regexp**, **tuple**, **array**, **unordered_** containers.
- Variadic templates
- ...