C++ 2011

Jacques Farré

Pompé sans vergogne sur la page web de B. Stroustrup :
http://www.stroustrup.com/C++11FAQ.html

et aussi d'autres nombreuses sources

C++ 2011 (objectifs)

- Maintien de la compatibilité avec C++ 99
- Plutôt ajouter des bibliothèques que des extensions au langage
- Donner de préference des solutions générales
- Satisfaire les débutants autant que les experts

  (est-ce de l'humour anglo-saxon ?)
- Augmenter la sécurité du typage
- Améliorer les performances, notamment pour les systèmes embarqués
- Correspondre aux besoins du monde réel

C++ 2011 (principales extensions)

- Initialisations et listes d'initialisation
- Références et pointeurs
- Templates variadiques
- Fonctions et Lambda-expressions
- Constructeurs hérités
- Boucles
- Sécurisation de la programmation concurrente
- ...
- Compiler (g++) avec l'option **`-std=c++0x`**

Extensions C++ 2011
auto

- Déclaration auto : la variable prend le type de sa valeur d'initialisation :

```
auto x = y + z;
```

  - bien agréable dans

```
template <typename T>
void F (const list<T>& l)
 for (auto i = l.begin();
           i != l.end();
           ++i) …
```

  au lieu de

```
for (typedef list<T>::iterator i = …
```

Extensions C++ 2011
decltype

- decltype : le type déclaré d'une expression

```
template <typename T, typename U>
void M (vector<T>& a, vector<U>& b) {
 typedef decltype (a[0] * b[0]) resultat;
 vector<resultat> r; int i;
 for (i = 0; i < a.size(); ++i)
   r[i] = a[i] * b[i];
```

Extensions C++ 2011
type résultat auto

- Comment déclarer le type de retour d'une fonction template dans le cas général ?

```
template <typename T, typename U>
??? mul (T x, U y) { return x * y; }
```

- Utiliser decltype ?

```
decltype(x*y) mul (T x, U y) { return x*y; }
```

ne fonctionne pas car types de x et y pas connus

- Solution retenue

```
auto mul (T x, U y) -> decltype(x * y)
   { return x * y; }
```

Extensions C++ 2011 expressions constantes

- Expressions constantes : aussi sur les valeurs d'objets

```
struct A {
 int x, y;
 constexpr A (int a, int b) : x(a), y(b) { }
};

constexpr A a[] = { A(0,0), A(1,1), A(2,2) };

constexpr int x = a[1].x;  // x = 1
```

Extensions C++ 2011 expressions constantes

- Expressions constantes :
  - permet d'avoir des expressions constantes plus générales, en particulier sur des types définis par l'utilisateur
  - garantit que l'initialisation sera faite à la compilation

```
enum Flags { good=0, fail=1, bad=2, eof=4 };

constexpr int operator|(Flags f1, Flags f2 {
  return Flags(int(f1)|int(f2));
}

constexpr int x = bad|eof;
```

Extensions C++ 2011
nullptr

- nullptr : 0 désigne jusqu'ici le pointeur nul et l'entier zéro
  - Risque d'ambiguïté

    ```
    void f(int); void f(char*);
    f(0); // = f(int)
    ```

    pour `f(char*)`, écrire `f((char*)0);`
  - maintenant

    ```
    f(0);                 // f(int)
    f(nullptr);           // f(char*)
    ```

Extensions C++ 2011
énumérations

- Énumérations fortement typées :
  - les noms restent dans l'espace du type
  - pas de conversion implicit vers int

```
enum Alert { green, yellow, red };
                                // traditional enum
enum class Color { red, yellow, green, blue };
                // scoped and strongly typed enum

Alert a = 7;  // error (as ever in C++)
Color c = 7;  // error, no int -> Color conversion
int a2 = red; // ok: Alert -> int conversion
int a3 = Alert::red; // error in C++98; ok in C++11
int a4 = blue;       // error: blue not in scope
int a5 = Color::blue; // error: no Color -> int conversion
Color a6 = Color::blue; // ok
```

Extensions C++ 2011 initialisations

- Initialisations rendues plus uniformes
  - Jusqu'ici seulement

```
double v[] = { 2, 3.45, 99.99 };
struct S { int x; double y, char* z} s
          = { 3, 0.5, "hello" };
```

notez qu'on peut écrire maintenant >>

  - Maintenant :

```
vector<double> v = { 2, 3.45, 99.99 };
map<string>,vector<int>> years = {
  { "Wilkes", {1913, 1945, 1951, 1967, 2000} },
  { "Ritchards", {1982, 2003, 2007} },
  { "Wheeler", {1927, 1947, 1951, 2004} };
```

Extensions C++ 2011 initialisations

- Ne pas confondre

```
vector<double> v1 (7);
// ok: v1 a 7 éléments construits par défaut

vector<double> v2 = {9};
// ok: v2 a 1 élément (de valeur 9)

vector<double> v2 {9};
// comme dessus

vector<double> v1 = 7;
// ko : le constructeur vector (int) est
// explicite (pas de conversion int → vector)
```

Extensions C++ 2011 initialisations

- Initialisation de membres de classe
  - Jusqu'ici, un membre de classe ne pouvait être initialisé que dans le constructeur
  - sauf les membres statiques
  - source d'incohérences si plusieurs constructeurs
- Maintenant, comme en Java

```
class A {
public:
  int a = 7;
…
};
```

L'initialisation par le constructeur a le dessus sur l'initialisation globale

```
A::A (int x) : a(x) { } //a=x
A::A() { }              //a=7
```

Extensions C++ 2011
listes de paramètres

- Liste de paramètres :

```
void f(initializer_list<int>);
f({1,2});
f({23,345,4567,56789});
f({}); // the empty list
f{1,2};// error: function call ( ) missing
```

- Les constructeurs et méthodes de la STL ont maintenant des listes de paramètres

```
years.insert({ {"Bjarne","Stroustrup"},
               {1950, 1975, 1985} } );
```

Extensions C++ 2011
override

- Override : le problème

```
class B {
  virtual void f();
  virtual void g() const;
  virtual void h(char);
  void k();  // pas virtuelle
};

class D : public B {
  void f();  // redéfinit B::f(); probablement OK
  void g();  // ne redéfinit pas B::g()
             // est-ce volontaire ?
  virtual void h(char); // redéfinit B::h(); voulu ?
  void k();  // ne redéfinit pas B::k()
             // (B::k() pas virtuelle)
             // est-ce volontaire ?
```

Extensions C++ 2011
override

- Override : la solution

```
class B {
 virtual void f();
 virtual void g() const;
 virtual void h(char);
 void k(); // pas virtuelle
};

class D : public B {
 void f() override;   // redéfinit clairement B::f()
 void g() override;   // ne redéfinit pas B::g()
                      // produit une erreur
 virtual void h(char); // redéfinit B::h()
                       // mais pas indiqué -> warning
 void k() override; // erreur : B::k() pas virtuelle
```

Extensions C++ 2011
final

- final : comme en Java

```
class B {
  virtual void f() final;
   // ne peut plus être rédéfinie dans les
   // classes héritées
  virtual void g();
};

class D : public B {
  void f();
   // erreur: B::f ne peut pas être redéfinie
  void g();    // OK
};
```

  - Mais ne serait-il pas mieux que f ne soit pas virtuelle ?

Extensions C++ 2011 default et delete)

- default et delete : par exemple

```
class X {
  const X& operator= (const X&) = delete;
     // on ne peut pas copier
  X (const X&) = default;
     // mais on peut construire par copie
     // avec le constructeur fourni par défaut
  ...
```

- Utile pour éviter les conversions non désirées

```
class X {
  X (double);
  X (int) = delete;
     // ne peut être construit avec un entier
  ...
```

Extensions C++ 2011
array

- Les tableaux C sont problématiques
- Les vector<T> sont un peu lourd
- Introduction des `array` dans la bibliothèque (**`#include <array>`**) comme compromis
  - tableaux de taille fixe, initialisables par liste,

```
array<int,6> a = { 1, 2, 3 };
             // les 3 derniers initialisés à 0
a[7] = 4; // pas de vérification de l'indice
int x = a[5];  // x = 0
int* p1 = a;   // erreur
int* p2 = a.data(); // ok
int l = a.size();
```

Extensions C++ 2011
unions généralisées

- Jusqu'ici, impossible d'avoir un membre d'union avec un constructeur (ou un destructeur ou un operator=) défini par l'utilisateur
- Autorisé maintenant, sous conditions : le membre de l'union ne doit pas avoir de fonction virtuelle, ne doit pas hériter, ne doit pas être une référence)
- Mais l'union n'a plus la fonction correspondante par défaut, l'utilisateur doit la préciser : ça se comprend, car le compilateur ne peut pas deviner ce qu'il faut faire

Extensions C++ 2011 conversions

- Conversion explicite :
  - actuellement, seuls les constructeurs peuvent exiger une conversion explicite
  - maintenant, les opérateurs de conversion aussi

```
class A {
  A (float);
  explicit A (int);
  operator float();
  explicit operator int();
};

A a = 3.14; A b (3);
A c = 5; // erreur
float f = a; int k = int (a);
int n = a; // erreur
```

Extensions C++ 2011 assertions statiques

- Jusqu'ici les assertions sont calculées à l'exécution
- Maintenant, si une expression est constante, l'assertation peut être calculée à la compilation :
  `static_assert (expression conastante, chaîne);`
- Par exemple

```
static_assert (sizeof(long) >= 8,
               "64-bit code generation required");
struct S { X m1; Y m2; };
static_assert (sizeof(S) == sizeof(X)+sizeof(Y),
                "unexpected padding in S");
```

Extensions C++ 2011
chaînes ”brutes”

- Chaînes ”brutes” : le problème des \
  ```
  string s = "\\w\\\\\\w";
  ```
  (donc `\w\\\w`, mais facile de se tromper)
  - maintenant, on peut avoir des chaînes sans traitement des \
    ```
    string s = R"(\w\\\w)";
    ```
  - et pour mettre un ” dans la chaîne : `R"("hi")"` (donc `"hi"`)
  - on peut choisir un délimiteur de début et fin : `R"#("("hi")")#"`
    (donc `"("hi")"`)
  - on peut spécifier le codage : `u8R"(éçà)"` (UTF8)

Extensions C++ 2011
littéraux utilisateur

- On peut définir de nouveaux littéraux grâce à des opérateurs de littéraux

```
constexpr complex<double> operator ""i
                                    (double d) {
   // partie imaginaire d'un complexe
   return {0,d};
}
auto z = 2 + 3i;  // complex (2,3)


string operator ""s (const char* p, int n) {
   return string (p, n);
}
auto x = "Hello!"s; // x string, pas char*
```

Extensions C++ 2011
littéraux utilisateur

- Littéraux définis par l'utilisateur :
  - Sur des dimensions physiques par exemple

```
constexpr longueur operator ""m  (double d) {
   return longueur (d);
}
constexpr longueur operator ""km  (double d) {
   return longueur (d * 1000);
}
longueur l = 12km + 5m;
```

  - Si le constructeur longueur(double) est privé, pas possible d'écrire `longueur l = 3;` (quelle unité ?)

    (les opérateur `""m` et `""km` doivent être amis en ce cas)

Extensions C++ 2011 boucles a la Java

- Appelées *range-for statements*

```
void f (vector<double>& v) {
  for (auto x : v) cout << x << '\n';
  for (auto& x : v) ++x;
}

for (const auto x : { 1,2,3,5,8,13,21 })
  cout << x << '\n';
```

- Valable avec tout type ayant des itérateurs

Extensions C++ 2011 délégation de construction

- Comme en Java

```
class X {
 int a; int b,
public:
  X(int x, int y) {
   if (0 < x && x < y && y < max) {
    a = x; b = y;
   }
   else throw bad_X(x); }
  X() : X {1, 2} { } // attention {} et pas ()
 ...
```

Extensions C++ 2011
constructeurs hérités

- Les membres d'une classe ne sont pas dans la même portée que ceux de la classe mère
  - ce qui pose parfois des problèmes (notamment pour les méthodes)

```
struct B {
  void f(double);
};
struct D : public B {
  void f(int);
};

D d;
d.f(4.5); // appelle f(int) avec 4 en argument!
```

Extensions C++ 2011 constructeurs hérités

- On peut s'en sortir en réinjectant la méthode dans la fille

```
struct B {
  void f(double);
};
struct D : public B {
  using B::f;
  void f(int);
};

D d;
d.f(4.5); // OK appelle B::f(double) avec 4.5
```

Extensions C++ 2011
constructeurs hérités

- Pourquoi ne pas le faire avec les constructeurs ?

```
struct B {
  B(int) { }
};

struct D : B {
  using B::B; // déclare implicitement D1(int)
  int x;
};
```

mais attention

```
void test() {
  D d(6); // problème : d.x pas initialisé
  D e;    // erreur : D1 n'a pas de constructeur
                      par defaut
```

On ne peut donc jouer à ce jeu que si les attributs de D sont initialisés en dehors du constructeur :

```
int x = 10;
```

Extensions C++ 2011
rvalue references

- Jusqu'ici il était possible d'avoir des références que sur les lvalue (donc des variables)

```
int x; int& r = x; const int& z = 0;
int& c = 0;
```

- Maintenant, on peut avoir des références (non constantes) sur les rvalue (donc des valeurs, possiblement temporaires)

```
X a; X f();
X& r1 = a; // bind r1 to a (an lvalue)
X& r2 = f(); // error: f() is an rvalue
X&& rr1 = f();// bind rr1 to temporary
X&& rr2 = a; // error: bind a is an lvalue
```

Extensions C++ 2011
move et copy

- À quoi cela peut-il bien servir ? Considérons par exemple

```
class A {
 …
 A(const A&);  // copy constructor
 A(A&&);       // move constructor
 const A& operator=(const A&); // copy assignment
 A& operator=(A&&);  // move assignment
};
```

Notez que le *move constructor* et le *move assignment* prennent des && non const

Ils peuvent donc éventuellement modifier/détruire leur argument

Extensions C++ 2011
move et copy

- Supposons que A implémente un arbre binaire

```
template <type T> class A {
 struct N { A* gauche, droite; T val; } *racine;

 // opérations coûteuses si sémantique de valeur
 A(const A&);  // copy constructor
 const A& operator=(const A&); // copy assignment

 // opérations peu chères
 // move constructor
 A(A&& a) { racine = a.racine; a.racine = 0; }
 // move assignment
 A& operator=(A&& a) {
   racine = a.racine; a.racine = 0; return *this;
 }
```

Extensions C++ 2011
move et copy

- Jusqu'ici on ne peut faire que

```
void swap(A& a, A& b) {
  A tmp(a); // nous avons donc 2 copies de a
  a = b;    // nous avons donc 2 copies de b
  b = tmp;  // nous avons donc 2 copies de b
}
```

- Or, quand a est copié dans tmp, on n'a plus besoin de b ; idem après a = b; d'où

```
  A tmp = move(a);    // peut détruire a
  a = move(b);        // peut détruire b
  b = move(tmp);      // peut détruire tmp
```

move est une fonction bibliothèque retournant une référence sur rvalue : donc on appelera les *move constructor* et *move assignment*, pas chers

Extensions C++ 2011
variadic templates

- Jusqu'ici, pour avoir une fonction avec un nombre variable de paramètres, il fallait passer par stdarg :

```
#include <stdarg.h>
int somme (int nb, ...) {
  va_list ap; int i; int s = 0;
  va_start(ap, nb);
  for (i=0; i<nb; i++) s += va_arg (ap, int));
  va_end(ap);
  return s;
}
...
int x = somme (3, 1, 2, 3);
int y = somme (6, -10, -5, 5, 12, -53, 67);
```

Extensions C++ 2011
variadic templates

- Maintenant, ... représente une liste avec une tête (un élément) et le reste (une liste)

```
template <typename T>   // spécialisation
T somme (T v) { // appelée avec nb = 1
  return v;
}
//
template <typename T, typename... Args>
T somme (T v, Args... a) {
  return v + somme (a...));
}
```

On instancie jusqu'à épuisement de la liste Args...

Extensions C++ 2011
variadic templates

- Mais attention, c'est piégieux

```
cout << somme (1, 3, 5 ) << '\n'; // 9
cout << somme (0.5, 3.5, 5.5 ) << '\n'; // 9.5
cout << somme (1, 3.5, 5 ) << '\n'; // 9 !!!
```

  car la fonction retourne le type du premier de la liste : **int**

```
cout << somme (3.5, 1, 5 ) << '\n';  // 9.5
```

- Les fonctions avec variadic template acceptent des variables et des expressions

```
double x = 3.5; int y = 5;
cout << somme (0.5, x + y, y - x) << '\n';
cout << somme (x, somme (1.5, x * y ) << '\n';
```

Extensions C++ 2011 variadic templates

- Variadic templates applicable aux classes

```
template<typename Head, typename... Tail>
class tuple<Head, Tail...>
                          : private tuple<Tail...> {
protected:
  Head h;
public:
  …
  const Head& head() const { return h; }
  Const tuple<Tail...>& tail() const { return *this; }
  tuple<Tail...>& tail() { return *this; }
};

tuple<string, vector<int>, double>
                          tt ("hi", {1,2,3}, 1.2);
```

Extensions C++ 2011 variadic templates

- Et pour construire et copier

```
tuple(const Head& v, const Tail&... vtail)
    : h(v), tuple<Tail...>(vtail...) { }

template<typename... V>
tuple(const tuple<V...>& t)
  : h(t.head()), tuple<Tail...>(t.tail()) { }

template<typename... V>
const tuple& operator=(const tuple<V...>& t){
  h = t.head(); tail() = t.tail(); return *this;
}
```

- Instancie jusqu'à épuisement des paramètres templates, donc ne pas oublier de définir :

```
template<> class tuple<> { };
```

Extensions C++ 2011 variadic templates

- Pénible de définir le type du tuple
  - Fonction de bibliothèque make_tuple

```
template<class... Types>
tuple<Types...> make_tuple(Types&&... t) {
  return tuple<Types...>(t...);
}

string s = "Hello";
vector<int> v = {1,2,3,4,5};
auto x = make_tuple(s,v,1.2);
```

- En fait, tuple est une classe de la bibliothèque (provenant de Boost)

Extensions C++ 2011
function et bind

- Fonctions de la bibliothèque (dans **<fonctional>**)
- bind permet de fixer certains paramètres d'un appel de fonction (remplace bind1 et bind2)

```
int f (int, bool, double);
auto ff = bind (f,_1, true, 1.2);
int x = ff(7);        // f (7,true,1.2);
```

où _1 indique le rang de l'argument de f dans l'appel via ff

```
auto frev = bind (f,_3,_2,_1);
int x = frev (1.2, true, 7); // f (7,true,1.2);
```

- Possible avec des fonctions surchargées, si on en donne la signature :

```
bind ((int(*)(int,bool,double))f,_1,true, 1.2);
```

Extensions C++ 2011
function et bind

- function permet de créer des objets-fonctions

```
function<float (int x, int y)> f;
// f est un objet fonction
// avec float operator() (int, int);

struct fdiv { // un object fonction
  float operator() (int x, int y) const {
    return ((float)x) / y;
  }
};
f = fdiv(); double d = f (5, 3);

float fd (int x, int y) {return ((float)x) / y;}
f = fd; d = f(4, 25);
```

Extensions C++ 2011
function et bind

- function fonctionne aussi avec des méthodes, mais il faut donner le type de this

```
struct X {
  int foo(int);
};
function<int (X*, int)> f;

f = &X::foo;  // pointeur sur méthode de X
X x;
int v = f(&x, 5);  // v = x.foo(5)

function<int (int)> ff = std::bind(f,&x,_1);
        // le premier argument de ff est donc x
v = ff(5);    // v = x.foo(5)
```

Extensions C++ 2011
lambda expressions

- Autre moyen de fabriquer des objets-fonction
- Il est souvent utile de passer une fonction en paramètre à une autre
  - Par exemple, ordre des valeurs pour un tri

```
template <typename T>
bool inf (int a, int b) { return a < b; }
vector<int> v;
sort (v.begin(), v.end(), inf<int>);
```

- Pourquoi dans un cas si simple définir une fonction ?
  - Utiliser des fonctions anonymes (lambdas)

```
sort (v.begin(),v.end(),
                    [](int a, int b) {return a<b;});
```

Extensions C++ 2011
lambda expressions

- On peut utiliser des variables locales dans la lambda

```
void f(vector<A>& v) {
vector<int> indices (v.size());

// générer un vecteur 0, 1, 2 … des indices
int count = 0;
generate (indices.begin(),indices.end(),
          [&count](){ return count++; });
// on passe count par référence à la lambda

// trier les indices selon l'ordre du champ x
de A : variables locales passées par référence
sort(indices.begin(), indices.end(),
  [&](int a, int b){ return v[a].x < v[b].x; });
```

Extensions C++ 2011
lambda expressions

- [...] est une ”capture” :
  - [&] indique que les variables locales sont passées par référence
  - [=] indique que les variables locales sont passées par valeur
  - [&v] ou [=v] indiquent qu'on capture uniquement v
- Règles d'écriture d'une lambda :
  - donner sa liste de capture : **`[=v, &r]`**
  - ses paramètres et leurs types : **`(int a, float b)`**
    son corps : **`{ return v[a] < ++r[b]; } )`**.
  - optionnellement le type de retour : **`-> bool`**
    (mais en général il peut être déduit du return)

Extensions C++ 2011 pointeurs (tirés de Boost)

- Pointeurs uniques :
  - Seul possesseur de l'objet qu'il pointe
    `unique_ptr<X> p(new X); // mais pas = new X`
    non copiable, mais ”movable”
- Pointeurs partagés (avec compte de références), le *garbage collector* économique : `shared_ptr<int> p(new int);`
- Pointeurs faibles : utilisés en collaboration avec les shared_ptr : `weak_ptr<int> w(p);`
  - Utilisés notamment pour gérer les cycles et les problèmes posés par la programmation concurrrente