

PARTIE VII

C++11

Bruno Bachelet
Christophe Duhamel

Extensions d'ordre général (1/8)

- Expressions constantes généralisées: `constexpr`
- Constante pointeur vide: `nullptr`
- Enumérations fortement typées: `enum class`
- Boucle «*foreach*»
- Littéraux personnalisés: `operator ""`
- Nouveaux types
 - Chaînes de caractères: `char16_t`, `char32_t`
 - Grands entiers: `long long int`
- Chaînes «brutes»: `R" (...) "`

- Expressions constantes généralisées
 - Nouveau mot-clé: `constexpr`
 - Garantit qu'une expression est statique (i.e. connue à la compilation)
 - Problème
 - `int getTaille(void) { return 5; }`
 - `int tab[getTaille()+7];` ⇒ erreur
 - Solution
 - `constexpr int getTaille(void) { return 5; }`
 - Peut s'appliquer aussi sur des variables
 - `constexpr double pi = 3.1415;`
 - `constexpr double pi2 = 2*pi;`

- Constante pointeur vide

- Problème avec «**NULL**»: ambiguïté pointeur ou entier

- `#define NULL 0`

- Nouveau mot-clé: `nullptr`

- Exemple

- `void f(char *);`

- `void f(int);`

- `f(0);` ⇒ appel «`f(int)`»

- `f(NULL);` ⇒ appel «`f(int)`»

- `f(nullptr);` ⇒ appel «`f(char *)`»

- Énumérations fortement typées
 - Limites des énumérations classiques
 - Valeurs définies en dehors du scope de l'énumération
 - Possibilité de comparer des valeurs de types différents
 - Nouvelle forme d'énumération
 - `enum class Enum1 { e1, e2, e3 };`
 - `std::cout << e1;` \Rightarrow erreur
 - `std::cout << Enum1::e1;` \Rightarrow OK
 - `enum class Enum2 { e1, e2, e3 };` \Rightarrow pas de conflit
 - `std::cout << (Enum1::e1 == Enum2::e1);` \Rightarrow erreur
 - Possibilité de spécifier le type sous-jacent (défaut = `int`)
 - `enum class Enum1 : unsigned int { e1, e2, e3 };`

■ Boucle «*foreach*»

- Parcourir simplement une collection

- Cibles

 - Tableaux classiques

 - Conteneurs de la bibliothèque standard

 - Utilisation des itérateurs via «**begin()**» et «**end()**»

- Exemples

 - ```
float t[10];
for (float & val : t) ++val;
```

  - ```
std::list<Voiture> liste;
```

 - ```
for (const Voiture & v : liste)
 std::cout << v << std::endl;
```

- Littéraux personnalisés
  - Possibilité de définir ses propres suffixes pour un littéral
    - Déclenche l'appel d'un opérateur (à définir)
    - Permet de produire un objet à partir d'un littéral
    - Fonctionne pour les entiers, flottants, caractères et chaînes
  - Exemple
    - `double operator "" _degre(long double angle)`  
`{ return angle/180*PI; }`
    - `double angle1 = 90.0_deg;`  
`double angle2 = PI/2;`  
  
`assert(angle1==angle2);`

## ■ Nouveaux types

### □ Chaînes de caractères

#### ■ Support UTF-8, UTF-16 et UTF-32

- `const char s[] = u8"I'm a UTF-8 string.";`
- `const char16_t s[] = u"I'm a UTF-16 string.";`
- `const char32_t s[] = U"I'm a UTF-32 string.";`

### □ Grands entiers

- «`long int`» ne garantit pas 64 bits
- «`long long int`» garantit maintenant au moins 64 bits
- Harmonisation avec la norme C99



- Chaînes «brutes» / *raw strings*
  - Parfois utile d'éviter les échappements de caractères
    - Chaîne contenant un script, une expression régulière...
  - Ajout d'un préfixe devant une chaîne: **R**
    - `"(\+|-)?[[:digit:]]+"`  $\Rightarrow$  «\» vu comme échappement
      - Solution: `"\\+|-)?[[:digit:]]+"`
    - **R**`"((\+|-)?[[:digit:]]+)"`  $\Rightarrow$  aucun échappement
      - Les parenthèses extérieures servent de délimiteurs
    - Cas particulier où «) "» apparaît dans la chaîne
      - Personnalisation possible des délimiteurs
      - **R**`"foo(R"((\+|-)?[[:digit:]]+))"foo"`

# Extensions pour les classes (1/10)

---

- Référence à un temporaire: `&&`
- Opérateurs de mouvement: *move constructor / assignment*
- Choix explicite des opérateurs par défaut: `default`, `delete`
- Réutilisation de constructeurs: `using`
- Contrôle de la redéfinition de méthode: `override`, `final`
- Initialisation par liste: `{...}`
- Opérateurs de conversion explicites: `explicit`

- Référence à un temporaire («*rvalue*»)
  - *rvalue* non modifiable en C++03
    - Seul type autorisé: `const T &`
    - Exemple: `string getNom(void);`
      - `const string & nom = getNom();` ⇒ OK
      - `string & nom = getNom();` ⇒ interdit
  - *rvalue* modifiable en C++11
    - Nouvelle syntaxe: `T &&`
    - Exemple
      - `const string && name = getNom();` ⇒ OK
      - `string && nom = getNom();` ⇒ OK
  - A quoi ça sert ?
    - Il existe des cas où l'on veut modifier un temporaire
    - Notamment, pour optimiser la copie d'objets

## ■ Opérateurs de mouvement

### □ Exploiter les temporaires $\Rightarrow$ deux nouveaux opérateurs

- *Move constructor*: `A(A && a)`

- *Move assignment*: `A & operator = (A &&)`

### □ Ils sont préférés aux opérateurs traditionnels

- Quand la valeur passée en argument est une *rvalue*

### □ Exemple: copie de vecteurs

- `v2 = trier(v1); // Retourne une copie`

- C++03: temporaire recopié  $\Rightarrow$  perte de temps

- C++11: mouvement possible  $\Rightarrow$  évite la copie

- Le vecteur temporaire peut être «dépouillé» de ses données

- Le vecteur cible pointe sur le tableau du vecteur temporaire

- Le vecteur temporaire pointe sur «`nullptr`»

## ■ Choix explicite des opérateurs par défaut

- Par défaut, une classe possède
  - Un constructeur par défaut
  - Un constructeur de copie
  - Un opérateur d'affectation
- C++03: pour empêcher l'utilisation ⇒ déclarer privé
- C++11: choix explicite ⇒ «**default**» ou «**delete**»
- Exemple

```
class NonCopiable {
 NonCopiable(void) = default;
 NonCopiable(const NonCopiable &) = delete;
 NonCopiable & operator = (const NonCopiable &) = delete;
};
```

# Extensions pour les classes (5/10)

## ■ Réutilisation de constructeurs

- Définition d'un constructeur à partir d'un autre  
⇒ évite la duplication de code

```
class A {
 protected:
 int val;

 public:
 A(int i) : val(i) {}
 A() : A(0) {}
};
```

- Héritage des constructeurs de la classe mère

- `class A { public: A(int i); ... };`
- `class B : class A { using A::A; ... };`
- `B b(5);` ⇒ OK

## ■ Contrôle de la redéfinition de méthode

- Lors de la redéfinition d'une méthode, une erreur est vite arrivée

- Exemple

- `class A { public: virtual void f(int); };`
- `class B : public A { public: virtual void f(double); };`
- Compile, mais «B::f» ne redéfinit pas «A::f»

- Mot-clé «**override**» ⇒ intention de redéfinition

- `class B : public A`  
    `{ public: virtual void f(double) override; };`
- Contrôle à la compilation ⇒ erreur

- Mot-clé «**final**» ⇒ pas de redéfinition

- `virtual void f(void) final` ⇒ pas de redéfinition possible de «f»
- `class A final` ⇒ héritage de «A» impossible

- Initialisation par liste (1/3)
  - C++03: possibilité d'initialiser des «agrégats» par liste
    - Agrégat = tableau ou classe avec restrictions
      - `int t[] = {7,8,9};`
    - Classe «agrégat» = attributs publics, pas de constructeur
      - `class Paire { public: int x; double y; };`
      - `Paire p = {3,7.0};`
  - C++11: généralisation à n'importe quelle classe
    - ```
class Paire {  
    private: int x; double y;  
    public: Paire(int a,double b) : x(a),y(b) {}  
};
```
 - `Paire p = {3,7.0};` ⇒ appel constructeur

■ Initialisation par liste (2/3)

□ Autres syntaxes

- `Paire p{3,7.0};`
- `Paire p({3,7.0});`
- `return {3,7.0};` ⇒ déduction du type par la signature de la méthode

□ Permet d'uniformiser l'initialisation de variables

□ Permet d'éviter certaines ambiguïtés de syntaxe

- Exemple: `Paire p();`
- Interprété comme fonction: `Paire (*)(void)`
- Et non pas comme variable !
- Levée de l'ambiguïté: `Paire p{};`

■ Initialisation par liste (3/3)

□ Possibilité de capter cette forme d'initialisation

- Représentation d'une liste sous forme d'objet
- Classe générique: `std::initializer_list<T>`
- A condition que les valeurs soient du même type

□ Construction à partir d'une liste

- ```
class Vecteur {
 ...
 Vecteur(std::initializer_list<int> liste)
 : taille(liste.size()), tab(new int[taille]) {
 std::copy(liste.begin(), liste.end(), tab);
 }
};
```
- ```
Vecteur v = {7,8,9};
```

- Opérateurs de conversion explicites
 - ❑ C++03: mot-clé «**explicit**» pour les constructeurs de conversion (i.e. avec un seul argument)
 - ❑ Oblige à demander une conversion
 - Evite des conversions automatiques peu souhaitables
 - Peut faciliter la compréhension d'un code
 - ❑ C++11: extension aux opérateurs de conversion

Extensions pour la généricité (1/9)

- *extern template*
- Inférence de type: `auto`, `decltype`
- Syntaxe alternative du retour de fonction: `->`
- Alias de *template*: `using`
- *Variadic template*
- Assertion statique: `static_assert`
- Plus d'ambiguïté avec la syntaxe «>>»
 - ❑ C++03: `vector<pair<int,double> >`
 - ❑ C++11: `vector<pair<int,double>>`

■ *extern template*

- Compilation séparée \Rightarrow plusieurs unités de compilation
- Une instance d'un générique peut être compilée plusieurs fois
 - Perte de temps à la compilation
- Mot-clé «**extern**» \Rightarrow empêcher la compilation d'une instance
 - **extern template class std::vector<int>;**
 - Instance non compilée dans l'unité
 - Attention: s'assurer qu'au moins une unité compile l'instance

■ Inférence de type (1/2)

- ❑ Déclarer le type d'une variable n'est pas toujours évident
 - Complicé à écrire (e.g. instantiation d'un *template*)
 - Polymorphisme statique \Rightarrow difficile de connaître le type d'un retour
- ❑ Alors que le compilateur peut le déduire
 - Contrôle des types à la compilation
 - Capable de détecter une erreur de type \Rightarrow capable de corriger
- ❑ Exemple

```
std::vector<int> v = { ... };  
? it = std::find(v.begin(), v.end(), 5);  
if (it != v.end()) *it = 0;
```

Extensions pour la généricité (4/9)

- Inférence de type (2/3)
 - Première possibilité: mot-clé «**auto**»
 - C++03: **auto** **int** **x**; \Rightarrow incorrect en C++11
 - C++11: **auto** **x** = ...; \Rightarrow déduction du type
 - **auto** = joker
 - Le programmeur laisse le compilateur déduire
 - **auto** **it** = **std::find**(**v.begin**(),**v.end**(),5);
 - Peut remplacer le retour d'une fonction
 - **auto** **f**(**int** **a**,**int** **b**) { **return** **a+b**; }
 - Peut remplacer tout ou partie d'un type
 - **auto** * **x**, **auto** & **x**, **const** **auto** & **x**...
 - **auto** **x** = **new** **auto**(5);

- Inférence de type (3/3)
 - ❑ Seconde possibilité: `decltype(expression)`
 - ❑ Représente le type d'une expression
 - Demande au compilateur de déduire le type
 - ❑ Exemple
 - `decltype(v.begin()) it;`
 - `it = std::find(v.begin(), v.end(), 5);`
 - ❑ Avec les deux approches, le type est connu à la compilation
 - Les contrôles de types sont donc préservés
 - ❑ Mais le type n'est pas explicite dans le code

- Syntaxe alternative du retour de fonction
 - Problème: type de retour placé avant les arguments
 - Déduction de type impossible
 - `template <typename A,typename B>`
`decltype(a+b) add(const A & a,const B & b)`
`{ return a+b; }`
 - Solution: placer le type de retour après les arguments
 - Nouvelle syntaxe: `->`
 - `template <typename A,typename B>`
`auto add(const A & a,const B & b) -> decltype(a+b)`
`{ return a+b; }`

■ Alias de *template*

- C++03: alias de type «*template*» impossible

- `template <typename T> typedef pair<int,T> paire_t;`

- C++11: nouveau mot-clé «*using*»

- `template <typename T> using paire_t = pair<int,T>;`

- Remplace «*typedef*»

- `typedef pair<int,double> paire2_t;` ⇒ *old style*

- `using paire2_t = pair<int,double>;` ⇒ *new style*

■ *Variadic template*

- Générique à paramètres variables
 - Liste des paramètres *templates* non fixée
 - A l'instar des arguments variables d'une fonction
- Exemple: collection de valeurs de types différents
 - Déclaration (partielle)
 - `template <typename... PARAMS> class Tuple;`
 - Instanciation
 - `Tuple<int,double,std::string> t;`
- Syntaxe simple pour instancier le générique
- Mais syntaxe peu intuitive pour écrire le générique
 - Nécessite une approche récursive (cf. métaprogrammation)

■ Assertions statiques

□ Solutions existantes

- `#error message` \Rightarrow interprétée par le préprocesseur
- `assert(expression)` \Rightarrow évaluée à l'exécution

□ Pas satisfaisantes pour les *templates*

- En particulier pour la métaprogrammation

□ Nouvelle solution: `static_assert(expression, message)`

- L'expression doit être constante (sinon utiliser «`assert`»)

□ Exemple

```
template <typename T> T f(const T & x, const T & y) {  
    static_assert(std::is_integral<T>::value,  
                  "f() doit recevoir des entiers");  
    ...  
}
```

- Facilitent la programmation fonctionnelle en C++
- Fonctions anonymes créées «à la volée»
 - `[] (int x,int y) { return x+y; }`
- Type de retour implicite
 - Déduction de type à partir de l'expression retournée
 - Autrement dit, type de retour = `decltype(x+y)`
- Type de retour explicite
 - `[] (int x,int y) -> int { return x+y; }`
 - Type de retour précisé après la déclaration
- Peuvent être utilisées comme foncteurs avec la STL
 - `std::find(v.begin(),v.end(),
[] (int x) { return x!=0; });`

Extensions de la *C++ Standard Library* (1/3)

■ *Multithreading*

- Création/manipulation de *threads*
 - `thread`, `thread::join()`...
- Synchronisation
 - `mutex`, `condition_variable`...

■ Expressions régulières

- Plusieurs normes dont POSIX
- `regex_search`, `regex_replace`...

■ Nombres aléatoires

- Générateurs de meilleure qualité que «`rand`»
 - `linear_congruential_engine`, `mersenne_twister_engine`...
- Distributions
 - `poisson_distribution`, `exponential_distribution`...

Extensions de la *C++ Standard Library* (2/3)

■ Tuples

- Collections d'objets de types hétérogènes

- `tuple<int,double,string> t = {13,2.7,"hello"};`

■ *Smart pointers*

- `unique_ptr`, `shared_ptr`, `weak_ptr`

- «`auto_ptr`» obsolète

■ *Type traits*

- `is_integral<T>`, `is_void<T>`, `is_array<T>...`

Extensions de la *C++ Standard Library* (3/3)

■ Conteneurs

□ Table de hachage

- Conteneur associatif sans tri
- `unordered_set`, `unordered_multiset`,
`unordered_map`, `unordered_multimap`

□ Tableau

- Conteneur représentant un tableau statique
- Classe générique: `array<T,N>`
- Interface standard: `size()`, `begin()`, `end()`...

□ Amélioration de l'interface

- Méthode «`insert`» avec *rvalue*
- Méthode générique «`emplace`» pour optimiser l'ajout