Le standard C++11

ENSIIE2 : Option LOA

Jean-Yves Didier

didier@ufrst.univ-evry.fr

Généralités

Que couvre la norme C++11 ?

- Le langage (*core language*) ;
- La bibliothèque standard du C++ (*C++ standard library*).

Historique

Phases de normalisation

1979 C with classes (Bjarne Stroustrup)

1983 C++

1998 ISO/IEC 14882 :1998 – C++98

2003 ISO/IEC 14882 :2003 – C++03

2007 ISO/IEC TR 19768 :2007 – C++TR1

2011 ISO/IEC 14882 :2011 – C++11 (août)

2014 C++14 ?

2017 C++17 ?

Support des compilateurs (1/2)

GCC

- Support très partiel de la version 4.3 à 4.6 (commutateur `-std=c++0x`) ;
- Support quasi complet depuis la version 4.7 (commutateur `-std=c++11`) ;
- Support complet depuis la version 4.8.1 (31/05/2013).

Clang

- Support très partiel de la version 2.9 à 3.0 ;
- Support quasi complet depuis la version 3.2 (commutateur `-std=c++11`) ;
- Support complet depuis la version 3.3 (17/06/2013).

Support des compilateurs (2/2)

Microsoft Visual C++

- Support très partiel de la version VC10 ;
- Support partiel des versions VC11 et VC12 (Visual Studio 2013).

Les principes de la nouvelle norme

- Une écriture plus simple ;
- Un contrôle amélioré du comportement du compilateur ;
- Une standardisation de techniques employées avant (dans `boost` par exemple) ;
- Le maintien de la rétro-compatibilité avec les précédentes normes C++.

Sémantique de déplacement (1/3)

Un exemple simple

```
class A { ... }
...
A x;
...
x = A();
```

Que fait exactement la dernière ligne ?

Sémantique de déplacement (2/3)

La méthode C++03 : l'affectation par copie

- A() créé un objet temporaire de type A ;
- Affectation par copie de l'objet temporaire vers x :
  - libérer les resources détenues par x précédemment ;
  - créer de nouvelles ressources pour x ;
  - copier le contenu des ressources de l'objet temporaire vers x ;
- Objet temporaire détruit.

Les limites

Si la ressource est coûteuse à créer et/ou à détruire, on l'a créé et détruite deux fois alors que l'on ne voudrait le faire qu'une fois : à la création et à la destruction de l'objet temporaire et déplacer le résultat dans x.

Sémantique de déplacement (3/3)

La solution : sémantique de déplacement

Consiste à transférer entre deux instances leurs états internes tout en minimisant le surcoût mémoire/temps de calcul.

Notions périphériques :

- *rvalue* et *lvalue* ;
- Constructeur par déplacement (*move constructor*) ;
- Affectation par déplacement (*move assignment*) ;
- Elision de copie (*copy elision*) ;
- Transfert parfait (*perfect forwarding*).

rvalue et *lvalue* (1/4)

En C

- *lvalue* : expression qui peut apparaître à gauche ou à droite d'une affectation ;
- *rvalue* : expression qui ne peut apparaître qu'à droite.

En C++

- *lvalue* : Expression se référant à un emplacement mémoire et permettant de récupérer cette adresse avec l'opérateur & ;
- *rvalue* : Expression qui n'est pas une lvalue ou valeur anonyme qui n'existe que durant l'évaluation d'une expression.

rvalue et *lvalue* (2/4)

Le nouvel opérateur &&

Indique que l'adresse de l'expression est à destination du compilateur uniquement : l'adresse ne peut pas être récupérée en utilisant l'opérateur & à l'exécution du programme.
Créé une référence vers une *rvalue* (*rvalue reference*).

Convertir une *lvalue* en *rvalue*

Utilisation de `std::move()` (header `utility`).

rvalue et *lvalue* (3/4)

Example de discrimination entre *lvalue* et *rvalue*

```
#include <iostream>
#include <utility>
int f() { return 42; }
int g(int& p) {
  std::cout << "using lvalue: " << p << std::endl;}
int g(int&& p) {
  std::cout << "using rvalue: " << p << std::endl;}

int main() {
  int i = 15;
  g(i);                // using lvalue: 15
  g(std::move(i));     // using rvalue: 15
  g(23);               // using rvalue: 23
  g(f());              // using rvalue: 42
  return 0; }
```

rvalue et *lvalue* (4/4)

Ordre de résolution en cas de surcharge

Paramètre	rval	const rval	lval	const lval	Priorité
`T&&`	×				4
`const T&&`	×	×			3
`T&`			×		2
`const T&`	×	×	×	×	1

Constructeur et affectation par déplacement (1/3)

Canevas

```
class C {
  public:
    // constructeur par copie
    C(const C& ref) { ... }
    // constructeur par déplacement
    C(C&& mov) { ... }
    // affectation par copie
    C& operator=(const C& ref) { ... }
    // affectation par déplacement
    C& operator=(C&& mov) { ... }
};
```

Constructeur et affectation par déplacement (2/3)

A noter

La disparition du mot-clé `const` dans le constructeur et l'affectation par déplacement (logique car *mov* est appelé à être modifié).

Retour à l'exemple de départ

`x=A();` Que fait une affectation par déplacement ?

La méthode C++11 : l'affectation par déplacement

- A() créé un objet temporaire de type A ;
- Affectation par déplacement de l'objet temporaire vers x :
  - échanger les ressources de l'objet temporaire et de x ;
- Objet temporaire détruit.

Constructeur et affectation par déplacement (3/3)

Les conditions d'application

C(C&& mov){ ... }

C& **operator**=(C&& mov){ ... }

mov doit être dans un état destructible et affectable à la fin de ces opérations.

Résolution par le compilateur

- Si paramètre est *rvalue* :
  - Si constructeur/affectation par déplacement disponible :
    - Appel de ce dernier.
- Par défaut, appel constructeur/affectation par copie.

L'élision de copie (*copy elision*) (1/5)

Elision : définition

- En langue : disparition d'une voyelle (généralement signalé par une apostrophe) ;
- L'élision de copie : optimisation du compilateur permettant d'éviter, sous certaines conditions, la copie d'un objet.

Exemple de code sujet à optimisation

```
int f() { int n=42; return n;}
void affiche(int i) { std::cout << i << std::endl; }
affiche(f());
```

L'élision de copie (*copy elision*) (2/5)

Compilation sans optimisation

- Appel de *f* : `int f() { int n=42; return n;}` :
  - Créé une zone sur la pile pour valeur de retour ;
  - Créé une zone sur la pile pour *n* ;
  - Affecte valeur à *n* ;
  - **Copie** *n* dans valeur de retour.
- Appel de *affiche* : `affiche(f())` :
  - Crée une zone sur la pile pour *i* ;
  - Crée une zone sur la pile pour valeur retour *f*() ;
  - **Copie** valeur retour *f*() dans *i* ;
  - Lance *affiche*().

L'élision de copie (*copy elision*) (3/5)

Se passer des opérations de copie (*copy elision*)

- Si une fonction retourne un objet local : optimisation de la valeur de retour (*return value optimization – RVO*) ;
- Si un objet à son initialisation ou une fonction prend en paramètre une entité temporaire.

Compilation avec optimisations

- Appel de *f* : `int f() { int n=42; return n;}` :
  - Créé une zone sur la pile pour valeur de retour ;
  - Affecte valeur à valeur retour ;
- Appel de *affiche* : `affiche(f())` :
  - Crée une zone sur la pile pour valeur retour de *f* ;
  - Lance *affiche*().

L'élision de copie (*copy elision*) (4/5)

Compilateur et élision de copie

- Optimisation non requise ;
- Optimisation autorisée même si production d'effets de bord !
- Comportement prévu dans la norme (sauf pour les exceptions).

Exemple d'effet de bord

```
#include <iostream> // à compiler avec et sans
int n=0;            // -fno-elide-contructors
struct C {          // (option g++ et clang)
  C() { }
  C(const C& ) { std::cout << ++n << std::endl; }
};
C f() { C c; return c; }
int main() { C a = f(); return 0; }
```

L'élision de copie (*copy elision*) (5/5)

Elision de copie et sémantique de déplacement

```
C faireQuelquechose() {
  C tmp;
  ...
  return tmp;
}
```

Comportement du compilateur :

- Si constructeur par copie ou par déplacement défini pour C, élision de copie possible ;
- Sinon appel du constructeur par déplacement (si existant) ;
- Sinon appel du constructeur par copie (si existant) ;
- Sinon erreur de compilation.

Transfert parfait – *perfect forwarding* (1/2)

Exemple

```
#include <iostream>
#include <utility>
void f(int& n) { std::cout<<"lv: "<<n<<std::endl; }
void f(int&& n){ std::cout<<"rv: "<<n<<std::endl; }

template<class T> void test_forward(T&& t) {
  f(t);       f(std::forward<T>(t));
}

int main() {
  int i = 1;
  test_forward(i); // affiche lv:1 puis lv:1
  test_forward(2); // affiche lv:2 puis rv:2
  return 0;
}
```

Transfert parfait – *perfect forwarding* (2/2)

Explication de texte

La fonction `test_forward (T&& t)` :

- Prend en argument une *lvalue* ou une *rvalue* ;
- A pour paramètre *t*, donc stockage dans une variable ;
- Donc *t*, dans *test_forward*, est une *lvalue* !

La fonction `std::forward`

- Déclarée dans l'en-tête `utility` ;
- Quand utilisée dans un template de fonction, transfert l'argument de la fonction exactement tel qu'il a été passé.

Contrôle des fonctions générées par défaut (1/3)

Compilateur et classes

Le compilateur génére (si pas spécifié explicitement) :

- Constructeur par défaut ;
- Constructeur par copie ;
- Affectation par copie ;
- Destructeur par défaut.

Nouveauté de C++11

Il est possible de contrôler le comportement de compilateur.

Contrôle des fonctions générées par défaut (2/3)

Cas du constructeur par défaut

- Seulement généré si pas d'autre constructeur défini ;
- Parfois, on voudrait avoir celui par défaut en plus :
  - Utilisation du modificateur `=default`.

Exemple d'utilisation du constructeur par défaut

```
#include <iostream>
struct A  {
  A() = default ; // A tester en commentant
  A(int i) { std::cout << i  << std::endl;}
};
int main() {
  A a;
  return 0;
}
```

Contrôle des fonctions générées par défaut (3/3)

Le modificateur =delete

Permet d'interdire :

- La construction/affectation par copie ;
- La conversion implicite de paramètres.

Exemple d'objet non copiable par construction/affectation

```
class A {
  public:
    A(const A& a)=delete ;
    A operator=(const A& a)=delete;
};
```

Constructeurs délégués et constructeurs par héritage (1/3)

Constructeurs délégués

- Surcharge de constructeur : parfois duplication de code ;
  - C++03 : créér une méthode appelée par les constructeurs ;
  - C++11 : les constructeurs peuvent s'appeler entre eux. Un constructeur délègue une partie de la construction de l'objet à un autre constructeur.

Exemple de constructeur délégué

```
class A {
    int nombre;
public:
    A(int n) { nombre=n ;}
    A() : A(42) {} // construction déléguée
};
```

Constructeurs délégués et constructeurs par héritage (2/3)

Constructeurs par héritage (inheriting constructors)

Idée : bénéficier des constructeurs de la classe mère directement.

Exemple de constructeur par héritage

```
struct A {
    A(int value);
};
struct B : A {
    using A::A;
};
int main() {
  B b(1);
  return 0;
}
```

Constructeurs délégués et constructeurs par héritage (3/3)

Constructeur par héritage : restrictions

- Importation de type tout ou rien ;
- Héritage multiple : import des constructeurs des classes de base possible si les constructeurs n'ont pas les mêmes signatures.

Contrôle de l'héritage et du polymorphisme

Modificateur `override`

Préviens la création accidentelle d'une surdéfinition à la place d'une rédéfinition.

Exemple d'utilisation d'`override`

```
struct A { virtual void fonction(float); };
struct B : A {
  virtual void fonction(int) override;
  // erreur à la compilation car pas une redéfinition.
};
```

Modificateur `final`

- Identique au mot-clé `final` en Java ;
- Préviens l'héritage de classe ou la redéfinition.

Initialisation des membres de classes

Initialisation de membres

- Possible désormais pour tous les membres ;
- Sauf spécification contraire dans un constructeur, valeur prise par la variable.

Exemple

```
class A {
public:
    A() {}
    A SomeClass(int nval) { val=nval; }
private:
    int val = 5;
};
```

L'inférence de type (1/2)

Définition

Capacité qui permet à compilateur de deviner le type d'une expression sans que ce dernier soit explicite dans le code source.

Possible en C++11 (dans une certaine mesure)

- Mot-clé **auto** : type à deviner ;
- Mot-clé **decltype** : typer une variable à partir du type d'une autre.

L'inférence de type (2/2)

Parcourir un vecteur en C++03

```
for (std::vector<int>::const_iterator itr=myvec.cbegin
    (); itr!=myvec.cend(); ++itr)
```

Parcourir un vecteur en C++11

```
for (auto itr=myvec.cbegin(); itr!=myvec.cend(); ++itr)
```

Les *templates* (1/4)

Templates externes

- Diminuent le temps de compilation en prévenant les compilations multiples du même *template* ;
- Indiquer au compilateur de ne pas instancier ici le *template* :

```
extern template class std::vector<MaClasse>;
```

Alias de *templates*

- Le `typedef` des *templates* : se fait à l'aide du mot-clé `using`.

```
template <typename T>
  using Dictionnaire = std::map< std::string, T >;
Dictionnaire<int> entiers;
entiers[ "un" ] = 1;
entiers[ "deux" ] = 2;
```

Les *templates* (2/4)

Templates variadiques

A l'instar des fonctions variadiques (par exemple *printf*), *template* qui prend une liste d'arguments variables.

Introduction de l'opérateur ...

Deux significations :

- A gauche d'un nom de paramètre :
  - Désigne un paquet de paramètres ;
  - Permet de lier 0 arguments ou plus à une fonction.
- A droite d'un nom d'argument, sépare les paramètres.

Accès aux paramètres

Se fait de façon récursive (à la compilation) !

Les *templates* (3/4)

Exemple de template variadique

```
#include <iostream>
void println() {
  std::cout << std::endl;
}
template<typename T,typename... Args>
void println(T arg1,Args... args) {
  std::cout << arg1 << "␣";
  println(args...);
}
int main() {
  println(1,"toto",2.0);
  println();
  println("hello","world");
  return 0;
}
```

Les *templates* (4/4)

Application des *templates* variadiques

Une réutilisation intensive dans C++11 :

- Introduction du type *tuple* dans la STL (sorte de structure templatisée) ;
- Introduction de la méthode `std::bind` et de l'objet `std::function` (voir dans la programmation fonctionnelle) ;
- Les initialisations uniformes ;
- Les nouvelles méthodes `emplace_back(...)` et `emplace(...)` dans les conteneurs de la STL.

Boucles ensemblistes

Boucles ensemblistes

Sucre syntaxique pour faciliter le parcours d'ensembles, de tableaux, de conteneurs de la STL (possédant les membres `begin()` et `end()`).

Exemple

```
int mon_tableau[5] = {1, 2, 3, 4, 5};
for (int &x: mon_tableau) {
    x *= 2;
}
for (int x:mon_tableau) {
    std::cout << "Valeur : " << x << std::endl;
}
```

Initialisation basée liste et Initialisation uniforme (1/3)

Initialisation basée liste

Introduction d'un nouveau template (variadique !) :

- De nom *initializer_list* déclaré dans l'en-tête éponyme ;
- Se parcourt comme une collection ;
- A utiliser en paramètre de constructeur ou de fonction ;
- Permet d'initialiser la classe ou la fonction en utilisant la notation ensembliste pour initialiser les tableaux (entre accolades).

Initialisation basée liste et Initialisation uniforme (2/3)

Exemple d'initialisation basée liste

```
#include <iostream>
#include <initializer_list>
struct S {
  S(std::initializer_list<int> l) {
    for(int x: l) sum+=l;
  }
  int sum=0;
};
int main() {
  S s={1,2,3,4,5,6};
  std::cout << "Somme :" << s.sum << std::endl;
  return 0;
}
```

Initialisation basée liste et Initialisation uniforme (3/3)

Initialisation uniforme

Syntaxe pour initialiser tout type avec la notation ensembliste.

Exemple d'initialisation uniforme

```
struct AltStruct {
  AltStruct(int x, double y) : x_{x}, y_{y} {}
private:
  int x_; double y_;
};
AltStruct var{2, 4.3};
```

Application de l'initialisation

En priorité, sur le constructeur avec `initializer_list`, puis par conversion implicite.

Fonctions *lambda* (1/5)

Fonctions anonymes

Fonction définie et probablement appelée sans être associée à un identifiant. Appelée aussi fonction *lambda*.

Syntaxe générale

```
[capture](parametres)->type-retour{corps}
[capture](parametres){corps}
[capture]{corps}
```

Exemple (trivial) d'utilisation d'une fonction anonyme

```
#include <iostream>
int main() {
  auto func=[]{std::cout<<"Hello world"<<std::endl;};
  func(); }// appel de la fonction
```

Fonctions *lambda* (2/5)

Exemples de fonctions *lambda*

```
// expression complète
[](int x, int y) -> int { int z = x + y; return z; }
// type de retour implicite lié à la clause return
[](int x, int y) { return x + y; }
// pas de clause return, type de retour = void
[](int& x) { ++x; }
// accès à une variable globale
[]() { ++global_x; }
// la même chose, les parenthèses peuvent être omises
[]{ ++global_x; }
```

Fonctions *lambda* (3/5)

Capture

Indique les variables définies dans le contexte créateur de la fonction anonyme (autres que globales) qui sont utilisées dans cette dernière (cf clôtures en programmation fonctionnelle).

Types de capture

[ ]	ne capture rien ;
[&]	capture par référence des variables (lecture/écriture) ;
[=]	capture par copie des variables (lecture seule) ;
[=,&a]	capture par copie des variables, sauf *a* par référence ;
[a]	capture par copie de *a* uniquement ;
[this]	capture le pointeur de la classe qui définit la fonction.

Fonctions *lambda* (4/5)

Exemple de capture et de clôture

```
#include <iostream>
// contexte appelant, pas de variable i définie
template<typename T> void call(T t) { t(); }

int main() {
  int i=0;
  // contexte créateur, i définie
  auto f=[&i]{i++;};
  call(f);
  // i modifié.
  std::cout << i << std::endl;
  return 0;
}
```

Fonctions *lambda* (5/5)

Fonction lambda et pointeurs de fonction

Compatibles ! Une fonction lambda peut-être stockée dans un pointeur de fonction.

Exemple : lambda stockée dans un pointeur de fonction

```
// déclaration d'un type de pointeur de fonction
typedef int (*func)();
// déclaration et initialisation
func f = [] () -> int { return 2; };
// appel de la fonction
f();
```

La génération d'adaptateurs à la volée (1/2)

bind : génération d'adaptateur

- Déclaré dans l'en-tête `functional` ;
- Permet de réordonner les paramètres ou de donner des valeurs préétablies.

Adapter une fonction

```
#include <cmath>
#include <functional>
#include <iostream>
int main() {
  auto cube=std::bind(pow,std::placeholders::_1,3);
  std::cout << cube(3) << std::endl;
  return 0;
}
```

La génération d'adaptateurs à la volée (2/2)

Les paramètres de `bind`

- Premier paramètre : fonction ou méthode de classe ;
- Liste d'arguments annexes :
  - Argument de type `std::ref` ou `std::cref` : passage d'une référence ;
  - Argument de type `std::placeholder::_x` : référence au xième argument de l'appel ;
  - Argument d'un autre type : directement inclu dans l'appel.

Pointeur de fonction templatisé (1/4)

function : pointeur de fonction généralisé

- Déclaré dans l'en-tête `functional` ;
- Peut copier, stocker, et invoquer n'importe quelle cible : fonctions, lambdas, expressions issues de *bind*, ...

Pointeur de fonction templatisé (2/4)

Exemple sur des fonctions classiques

```
#include <functional>
#include <iostream>

void printNum(int i) {std::cout<<i<<std::endl;}

int main() {
  std::function<void(int)> f_ = printNum;
  std::function<void()> f_42 = [](){printNum(42);};
  std::function<void()> f_31337=std::bind(printNum
      ,31337);
  f_(-9);
  f_42();
  f_31337();
  return 0;
}
```

Pointeur de fonction templatisé (3/4)

Exemple sur une méthode de classe

```
#include <functional>
#include <iostream>
struct Foo {
  Foo(int n) { num=n; }
  void printAdd(int i){std::cout<<num+i<<std::endl;}
  int num;
};
int main() {
  std::function<void(Foo&,int)> f_add_display = &Foo::
      printAdd;
  Foo foo(314159);    // méthode d'un objet.
  f_add_display(foo, 1);
  return 0;
}
```

Pointeur de fonction templatisé (4/4)

Exemple sur les captures

```
#include <iostream>
#include <functional>
void call(std::function<void()> t) { t(); }
int main() {
   int i=0;
   auto f=[&i]{i++;};
   call(f);
   std::cout << i << std::endl;
   return 0;
}
```

Le pointeur null (1/2)

Notations introduites

nullptr spécification d'un pointeur NULL ;

NULL macro définissant un pointeur NULL ;

nullptr_t type du pointeur NULL.

Un type pour les pointeurs null ? ! ?

Quel usage ?

Le problème

```
void f(int* i)    { if (i) *i ++ ; }
void f(double* d) { if (d) *d += 0.1; }
...
f(nullptr); // quelle fonction appeler ?
```

Le pointeur null (2/2)

Lever l'ambigüité

```
void f(int* i)    { if (i) *i++; }
void f(double* d) { if (d) *d += 0.1; }
void f(std::nullptr_t) {
    std::cout << "null" << std::endl;
}
...
f(nullptr); // 3e fonction appelée.
```

Pointeurs intelligents (*smart pointers*)

Définition

Type abstrait de données qui simule le comportement d'un pointeur tout en lui adjoignant des fonctionnalités supplémentaires telles que la gestion de la mémoire ou la vérification des bornes.

Les *smart pointers* en C++

3 categories (définies dans le *header* *memory*) :

- unique_ptr : Maintient un objet et un pointeur vers celui-ci de manière unique ;
- shared_ptr : Pointeur partagé par un mécanisme de compteur de références ;
- weak_ptr : Pointeur vers une donnée non possédée ;

unique_ptr (1/2)

unique_ptr : maintenir un objet et son pointeur

- Gére la durée de vie d'un objet créé par allocation dynamique ;
- Ne peut-être copié ou affecté par copie ;

Membres principaux

reset	remplace l'objet géré ;
release	abandonne la possession de l'objet et retourne un pointeur vers ce dernier (*get* retourne *nullptr* après) ;
get	retourne un pointeur vers l'objet possédé ;
operator*	déréférence le pointeur vers l'objet possédé ;
operator->	idem.

unique_ptr (2/2)

Exemple d'utilisation

```
#include <memory>
#include <iostream>
struct A {
  A() { std::cout << "a+" << std::endl; }
  ~A(){ std::cout << "a-" << std::endl; }
};
struct B { // pas de destructeur pour désallouer a.
  std::unique_ptr<A> a;
  B() { a.reset(new A()); }
};
int main() {
  B b;
  return 0;
}
```

shared_ptr (1/4)

shared_ptr : le mécanisme de compteur de références

- A chaque copie : incrémentation du compteur de références ;
- A chaque destruction d'une copie : décrémentation du compteur de références ;
- Lorsque le compteur est à 0, destruction de l'objet référencé.

Membres principaux

Cf membres de *unique_ptr*

use_count donne le nombre références communes à l'objet ;

unique indique si le *smart-pointer* est le seul à maintenir une référence à l'objet ;

shared_ptr (2/4)

Fonctions associées

Définie dans l'en-tête `<memory>` :

- `shared_ptr<T> make_shared(Args&&... args)`
  - Construit un pointeur intelligent ;
  - Utilise comme paramètres les paramètres du constructeur ;
  - Optimisation de `std::shared_ptr<T> p(new T(Args...))` ;
  - Evite des fuites mémoire si la construction de l'objet de type T est susceptible de lancer une exception.
- `shared_ptr<T> allocate_shared( const Alloc& alloc, Args... args );`
  - Comme précédemment, un allocateur en plus (fonction à laquelle on délègue la construction de l'objet).

shared_ptr (3/4)

Exemple d'utilisation

```
#include <memory>                                    //affichage:
#include <iostream>                                  // a+
struct A {                                           // sh1:2
  A() { std::cout << "a+" << std::endl; }            // sh2:2
  ~A(){ std::cout << "a-" << std::endl; }            // sh1:1
};                                                   // a-

int main() {
  std::shared_ptr<A> sh1(new A());
  { std::shared_ptr<A> sh2(sh1);
    std::cout<< "sh1:"<< sh1.use_count()<< std::endl;
    std::cout<< "sh2:"<< sh2.use_count()<< std::endl; }
  std::cout<< "sh1:"<< sh1.use_count()<< std::endl;
  return 0;
}
```

shared_ptr (4/4)

Le problème des références circulaires

```
#include <memory>       // affiche seulement a+
#include <iostream>     // a n'est donc pas détruit !!!
struct B;
struct A {
  A() { std::cout << "a+" << std::endl; }
  ~A(){ std::cout << "a-" << std::endl; }
  std::shared_ptr<B> b;
};
struct B { std::shared_ptr<A> a; };
int main() {
  std::shared_ptr<A> x(new A());
  x->b.reset(new B);
  x->b->a = x;
  return 0;
}
```

weak_ptr (1/3)

weak_ptr : gestion d'une référence temporaire

- Utilisé pour contrer le problème des références circulaires ;
- Gére une référence, n'agit pas sur le compteur de références ;
- Scrute le compteur de références pour vérifier si l'objet est encore alloué ;
- Pointeur utilisable seulement si *weak_ptr* réclame le contrôle.

Membres principaux

Cf membres de *shared_ptr* sauf pour operator* et operator-> (non définis).

expired booléen qui indique si l'objet pointé est désalloué ;

lock créé un *shared_ptr* temporaire pour travailler dessus.

weak_ptr (2/3)

Exemple d'utilisation

```
#include <iostream>    // affichage
#include <memory>      // a+
struct A {  ...  };    // pa alloue
std::weak_ptr<A> pa;   // a-
void f() {             // pa desalloue
  if (auto spt = pa.lock()) {
    std::cout << "pa alloue" << std::endl; } else {
    std::cout << "pa desalloue" << std::endl; }
}
int main() {
    { auto sp = std::make_shared<A>();
      pa = sp; f();
    }
    f();
}
```

weak_ptr (3/3)

Contrer les références circulaires

```
#include <memory>          // affiche
#include <iostream>        // a+
struct B;                  // a-
struct A {                 // fuite colmatée !
  A() { std::cout << "a+" << std::endl; }
  ~A(){ std::cout << "a-" << std::endl; }
  std::shared_ptr<B> b;
};
struct B { std::weak_ptr<A> a; };
int main() {
  std::shared_ptr<A> x(new A());
  x->b.reset(new B);
  x->b->a = x;
  return 0;
}
```

Concurrence

C++11 le propose enfin !

- Création de threads ;
- Gestion des exclusions mutuelles ;
- Gestion des opérations atomiques.

La classe *thread* (1/3)

Définie dans l'en-tête *thread*.
Permet de créer un nouveau fil d'exécution fonctionnant en parrallèle par le biais d'une invocation de fonction.

Membres principaux

operator=	Affectation par déplacement ;
joinable	vérifie si le thread est dans le même espace d'exécution ;
get_id	retourne l'identifiant du thread ;
join	attends la fin de l'exécution du thread ;
detach	permet au thread de s'exécuter indépendemment du programme principal.

La classe *thread* (2/3)

L'espace de nommage *this_thread*

Permet d'accéder ou d'influer sur le thread courant :

- yield : Démarre un réordonnancement des threads ;
- get_id : Donne l'identifiant du thread courant ;
- sleep_for : Endort le thread courant pour une période déterminée.

Implémentation en cours de stabilisation

Fait encore appel dans une large mesure aux threads POSIX.
Ne pas oublier le commutateur `-pthread` avec g++.

La classe *thread* (3/3)

Exemple d'exécution de thread

```
#include <iostream>
#include <thread>
void f(int n,int m) {
  for (int i=0; i< n; i++) {
    std::cout << i << "/" << n << std::endl;
    std::this_thread::sleep_for(std::chrono::
        milliseconds(m); }
}
int main() {
  std::thread t1(f,10,10);
  std::thread t2(f,5,30);
  t1.join();
  t2.join();
  return 0;
}
```

Les exclusions mutuelles (1/4)

Plusieurs types

Primitives de synchronisation utilisées pour protéger des données accédées simultanément par plusieurs threads.

mutex	Exclusion mutuelle standard ;
timed_mutex	Exclusion mutuelle pour laquelle la durée de réclamation est en temps fini ;
recursive_mutex	Exclusion mutuelle réentrante ;
recursive_timed_mutex	Combinaison des deux précédentes.

Les exclusions mutuelles (2/4)

Membres principaux

lock	vérouille le mutex, bloquant ;
unlock	dévérouille le mutex ;
try_lock	tentative de vérouillage non bloquante. Retourne vrai si réussi ;
try_lock_for	tentative de vérouillage bloquante pendant une certaine durée. Retourne vrai si réussi, faux si la durée a expirée.

Les exclusions mutuelles (3/4)

Exemple d'exclusion mutuelle (1/2)

```
#include <thread>
#include <iostream>
#include <cstdlib>
#include <mutex>
int sum = 0, tmp = 0;
std::mutex m;

void add(int n) {
  //m.lock(); // A decommenter
  tmp = n;
  std::this_thread::sleep_for(
    std::chrono::milliseconds(std::rand()%30));
  sum+=tmp;
  //m.unlock(); // A decommenter
}
```

Les exclusions mutuelles (4/4)

Exemple d'exclusion mutuelle (2/2)

```
void loop(std::string p) {
  for(int i=0; i<10; i++) {
    std::this_thread::sleep_for(
      std::chrono::milliseconds(std::rand()%30));
    add(i);
    std::cout << p << "(" << i << ")" << std::endl; }
}
int main() {
  std::srand(std::time(0));
  std::thread t1(loop,"t1");
  std::thread t2(loop,"t2");
  t1.join();
  t2.join();
  std::cout << "total " << sum << std::endl;
  return 0;
}
```

Le support de l'atomicité

Les objets de type atomique

Objets libres de toute situation de compétition : si un thread écrit dans l'objet pendant qu'un autre thread le lit, le comportement est conforme.
Basé sur la spécialisation du template `std::atomic`.
Exemple : std :: atomic<**bool**>
Disponible pour tous les types de base.

Un nouveau standard vaste

Parmi les nouveautés non traitées

- Encodage des chaînes de caractères, unicode ;
- Exceptions ;
- Conteneurs STL : *arrays*, *tuple*, tables de hashage ;
- Type de base : types entiers étendus, unions généralisés, enums fortement typés ;
- Templates : SFINAE étendu (*Substitution failure is not an error*) ;
- Concurrence : API asynchrone.

Un aperçu

A vous de voir ce que vous pouvez en faire.

Tester C++11 sur la YaKa

GCC 4.7.2

Procédure :

- Sourcer : `/usr/setup/gcc-47.bash` ;
- Afficher le script pour voir où il est installé exactement et le `LD_LIBRARY_PATH` nécessaire.

Pour aller plus loin...

- Documentation complète de l'API et des éléments de langage : http://en.cppreference.com
- La FAQ de Bjarne Stroustrup sur C++11 : http://www.stroustrup.com/C++11FAQ.html
- La page wikipedia anglophone (surtout pas la française !) : http://en.wikipedia.org/wiki/C++11
- Support comparé de C++11 par les compilateurs courants : http://wiki.apache.org/stdcxx/C++0xCompilerSupport