Le C++14 en détail

Auteurs : Kje, gbdivers, lmghs, germinolegrand

Dernière modification : 16 octobre 2014

Zeste de savoir

Table des matières

1 Un peu d'histoire **3**

2 Les nouvelles fonctionnalités de base **4**
2.1 Littéraux binaires (N3472) 4
2.2 Variables templates (N3651) 5
2.2.1 Remarque sur Clang (testé avec Clang 3.4.2) 6
2.3 Séparateurs de chiffres (N3781) 7
2.4 Dépréciation de fonctionnalités (N3760 et N3924) 7

3 Les nouvelles fonctionnalités sur les fonctions **10**
3.1 Relâchement des contraintes sur constexpr (N3652) 10
3.2 Généralisation de l'inférence de type 11
3.2.1 Rappels sur l'inférence de types en C++ 11
3.2.2 Utilisation de decltype(auto) 12
3.2.3 Uniformisation des syntaxes 13
3.2.4 Simplification des règles d'utilisation de -> 13
3.2.5 les fonctions lambda génériques (N3649) 14
3.3 Expression de captures dans les lambda (N3648) 14
3.3.1 Limitations des captures dans les lambda 14
3.3.2 Les approches possibles 15
3.3.3 Les nouvelles captures du C++14 16

4 Modifications sur la bibliothèque standard **17**
4.1 De nouveaux suffixes pour les littéraux (N3531) 17
4.2 Make_unique 18

5 Les autres apports du C++14 **20**

6 Et maintenant ? **21**
6.1 Support du C++14 22

7 Sources **23**
7.1 Pour aller plus loin 23

1 Un peu d'histoire

Le lundi 18 août, Herb Sutter[1] a annoncé sur le site du comité de standardisation du langage C++, que le nouveau standard du langage C++ avait été unanimement accepté et que la publication officielle allait être effectuée. Ce nouveau standard, d'un des langages de programmation les plus populaires et utilisés au monde[2], arrive trois ans après le C++11 (`ISO/IEC 14882 :2011` pour les intimes), dernier standard en date du langage.

Le C++ a été créé dans les années 1980 par Bjarne Stroustrup au sein du laboratoire Bell d'A&T. Au début dédié à l'ajout de classes dans le C[3], de nombreuses fonctionnalités lui sont introduites au fur et à mesure de son utilisation grandissante et a abouti en 1998 à la première normalisation ISO, le C++98 (`ISO/CEI 14882 :1998`). Après des modifications relativement mineures en 2003 (`ISO/CEI 14882 :2003`), une petite révolution arrive difficilement à terme en 2011 avec la ratification de l'`ISO/IEC 14882 :2011`, dite C++11. Cette version est la première évolution majeure de C++ depuis sa création et fait suite à une longue gestation.

Suite à la sortie du C++11, le comité de normalisation en charge de définir le langage s'est réorganisé pour faciliter les futures évolutions du langage et la bibliothèque standard. Au lieu de proposer une nouvelle norme après plusieurs années, l'évolution du langage sera plus dynamique :

- Au lieu d'avoir toutes les discussions sur les évolutions du langage qui sont discutées et décidées par le comité de normalisation au complet, plusieurs groupes d'études (SG, study groups) et de travail (WG, working groups) sont chargés d'étudier les différentes problématiques liées à un domaine et faire les propositions au comité de normalisation.
- Chaque proposition fait l'objet d'un *proposal* (identifié par la lettre N et un nombre à 4 chiffres) comme avant, mais au lieu d'attendre une nouvelle norme (ce qui prend du temps), les *proposals* peuvent être rassemblés par thématique et proposés comme « norme intermédiaire » (TS, technical specification). Ces TS peuvent être implémentés par les compilateurs, sans attendre la normalisation finale. Plusieurs TS sont actuellement en phase de finalisation (notamment File System, Networking, Concepts, on peut voir leur avancement sur isocpp.org/std/status). L'ensemble des TS formera une future norme (C++1z), probablement en 2017.

Le brouillon de normalisation de la prochaine version (C++14) a donc été approuvé cette semaine et deviendra officiellement une norme ISO d'ici quelques semaines, sous le nom de `ISO/IEC 14882 :2014` après quelques dernières corrections typographiques.

1. Herb Sutter est un expert reconnu du langage C++, auteur de plusieurs ouvrages sur ce sujet et secrétaire du comité de standardisation du langage C++ à l'ISO.
2. 4ème selon le dernier Index TIOBE d'août 2014, 5ème selon la dernière mesure de PYPL d'août 2014, 2ème selon moi ;)
3. il était d'ailleurs au début nommé *C with classes*.

2 Les nouvelles fonctionnalités de base

Le C++14 est une mise à jour mineure du langage en rien comparable aux changements importants introduits par le C++11. Il s'agit en réalité surtout de corriger des imprécisions dans la norme et l'ajout de quelques fonctionnalités oubliées précédemment, mais utiles en pratique. Aucun gros changement donc, mais nous allons vous présenter un petit résumé des principales nouveautés (le plus accessible possible).

2.1 Littéraux binaires (N3472)

En C++ un nombre entier peut être écrit sous différentes bases, en utilisant un préfixe composé du caractère 0 et d'un spécificateur optionnel pour la base à utiliser (rien pour l'octal, x ou X pour l'hexadécimal) :

```
1    int a = 42;    // Base décimal (10)
2    // ou int const a { 42 };   // bracket-initialization
3    // ou auto const a = 42;    // inférence de type
4
5    int b = 052;   // Base octal (8)
6    int c = 0x2A;  // Base hexadécimal (16)
```

Avec le C++14 (N3472), en utilisant le spécificateur b ou B :

```
1    int d = 0b00101010;           // Base binaire (2)
2    std::cout << d << std::endl;  // affiche 42
```

Pour rappel, il est possible d'afficher les nombres entiers avec `std : :cout` selon différentes bases, en utilisant des spécificateurs de format : `std : :oct` pour afficher en base octale (8), `std : :dec` pour la base décimale (10) et `std : :hex` pour la base hexadécimale (16). Ajoutons à cela le spécificateur `std : :showbase` pour afficher la base utilisée pour l'affichage.

Pour afficher la représentation binaire d'un nombre, il est possible d'utiliser la classe `std : :bitset` :

```
1    #include <iostream>
2    #include <bitset>
3
4    int main() {
5        unsigned char d = 0b00101010;
6        std::cout << "0b" << std::bitset<sizeof(d)*8>(d);
```

```
        // affiche 0b00101010
    }
```

2.2 Variables templates (N3651)

Pour définir une constante en C++, il est classique d'utiliser le mot-clé `const` (« constant ») dans le type d'une variable. On retrouve également parfois une ancienne syntaxe héritée du C, utilisant le préprocesseur et la directive `#define`.

```
    double const pi = 3.1415926535897932385;
    #define pi = 3.1415926535897932385
```

La syntaxe avec `#define` est dépréciée depuis de nombreuses années, elle ne permet pas au compilateur de vérifier la compatibilité des types des variables et peut poser des problèmes de remplacement des identificateurs dans le code (notamment parce qu'elles ne respectent aucune règle de portée).

Le C++14 introduit une nouvelle syntaxe pour déclarer les constantes (N3651). Cette syntaxe étant la plus générique, elle doit être préférée par défaut aux autres syntaxes.

```
    template<typename T>
    constexpr T pi = T(3.1415926535897932385);

    auto const piD = pi<double>;
    auto const piF = pi<float>;
    auto const piI = pi<int>; // ici piI == 3

    template<class T>
    T circle_area(T r) {
        return pi<T> * r * r;
    }
```

Cette syntaxe est également utilisable avec des types plus complexes, comme les classes. Par exemple, avec std : :complex :

```
    template<typename T>
    constexpr std::complex<T> i = std::complex<T>(0, 1);
```

Pour cela, il faut initialiser l'objet avec un constructeur constexpr (ce qui est le cas de std : :complex). Cette syntaxe est utilisable avec les types que vous créez.

```
    class MyClass {
    public:
        constexpr MyClass(int value);
    };

    template<typename T>
    constexpr MyClass<T> my_const = MyClass<T>(123);
```

Autre exemple, pour créer une liste de valeurs dans différents types de conteneurs.

```
template<template<typename...> class Seq>
Seq<int> primes = { 1, 2, 3, 5, 7, 11, 13, 17, 19 };

auto vec = primes<std::vector>;
auto list = primes<std::list>;
```

Toutes les règles classiques des templates s'appliquent aussi aux variables templates. En particulier, il est possible de spécialiser ses déclarations. Par exemple, on peut utiliser cette syntaxe pour définir des constantes basées sur des classes de traits :

```
template<typename T, typename U>
constexpr bool is_same = std::is_same<T,U>::value;

bool t = is_same<int,int>;   // true
bool f = is_same<int,float>; // false
```

En utilisant la spécialisation de template, il est possible d'utiliser la syntaxe suivante :

```
template<typename T, typename U>
constexpr bool is_same = false;

template<typename T>
constexpr bool is_same<T,T> = true;
```

Par défaut, il est donc préférable d'utiliser une syntaxe qui est la plus générique.

2.2.1 Remarque sur Clang (testé avec Clang 3.4.2)

Le code suivant :

```
#include <iostream>

template<typename T>
constexpr T pi = T(3.1415926535897932385);

template <class T>
T area(T r) {
    return pi<T> * r * r;
}

int main() {
    std::cout << "pi: " << area(1.0) << std:: endl;
}
```

produit une erreur de compilation :

```
main.cpp:4:13: warning: variable 'pi<double>' has internal
linkage but is not defined [-Wundefined-internal]
constexpr T pi = T(3.1415926535897932385);
            ^
main.cpp:8:12: note: used here
    return pi<T> * r * r;
           ^
1 warning generated.
/tmp/main-7b895c.o: In function `double area<double>(double)':
main.cpp:(.text._Z4areaIdET_S0_[_Z4areaIdET_S0_]+0xe):
undefined reference to `_ZL2piIdE'

clang: error: linker command failed with exit code 1 (use -v
to see invocation)
```

2.3 Séparateurs de chiffres (N3781)

Lors de la définition de grands nombres, il est parfois difficile de s'y retrouver devant beaucoup de chiffres. Ainsi les Américains ont tendance à utiliser la virgule comme séparateur de milliers (tandis que les Français utilisent le point). Un milliard s'écrit alors `1,000,000,000` (ou `1.000.000.000`) qui est plus facile à lire que `1000000000`. L'ajout de cette possibilité a longtemps été demandé, mais est apparu difficile à finaliser en raison des risques de conflits de syntaxe et des disparités selon les pays. La solution retenue (N3781) a donc été le symbole de citation simple `'` :

```
1   const int million = 1'000'000'000;
```

Le séparateur de chiffres peut s'utiliser avec les littéraux binaires, pour simplifier l'écriture :

```
1   const int million = 0b0000'0000'1010'1010;
```

2.4 Dépréciation de fonctionnalités (N3760 et N3924)

Chaque compilateur propose une syntaxe spécifique pour définir des extensions non standards :

- pour GCC et Clang : `__attribute__((...))` ;
- pour MSVC : `__declspec(...)`.

Cette syntaxe permet en particulier de noter qu'une fonctionnalité ne doit plus être utilisée (elles sont encore disponibles dans la norme actuelle, mais rien ne garantit qu'elles le seront encore dans la prochaine).

Table 2.1: Dépréciation sans message

Compilateur	Syntaxe
GCC et Clang	`__attribute__((deprecated)) int a;`
Visual Studio	`__declspec(deprecated) int a;`

Table 2.2: Dépréciation avec message

Compilateur	Syntaxe
GCC et Clang	`__attribute__((deprecated("message"))) int a;`
Visual Studio	`__declspec(deprecated("message")) int a;`

Pour homogénéiser les syntaxes, le C++11 a introduit la notion d'attributs, qui permet de définir de telles extensions non standards. Dans un premier temps, seuls les attributs `[[noreturn]]` et `[[carries_dependency]]` ont été définis dans la norme.

Le C++14 (N3760) introduit l'attribut `[[deprecated]]` pour n'avoir qu'une seule méthode pour déprécier des fonctionnalités. Il est possible d'utiliser cet attribut avec ou sans message :

```
1    [[deprecated]]
2    [[deprecated("message")]]
```

Cela permet de générer un avertissement (warning) lors de la compilation, avertissant l'utilisateur qu'il ne doit plus utiliser ces fonctionnalités :

```
1    [[deprecated]] void f() {}
2    [[deprecated("cette fonction est dépréciée")]] void g() {}
3
4    int main() {
5        f();
6        g();
7    }
```

Le code précédent génère les deux avertissements suivants (Clang 3.4) :

```
main.cpp:7:5: warning: 'f' is deprecated [-Wdeprecated-declarations]
    f();
    ^
main.cpp:3:21: note: 'f' has been explicitly marked deprecated here
[[deprecated]] void f() {}
                    ^
main.cpp:8:5: warning: 'g' is deprecated: cette fonction est
dépréciée [-Wdeprecated-declarations]
    g();
    ^
```

```
main.cpp:4:56: note: 'g' has been explicitly marked deprecated here
[[deprecated("cette fonction est dépréciée")]] void g() {}
                                                     ^

2 warnings generated.
```

Pour le moment, peu de fonctionnalités sont dépréciées dans le standard. Seule la fonction `std : :random_shuffle` est explicitement noté comme dépréciée dans N3924. La raison est que l'algorithme utilisé dans les anciennes fonctions de génération de nombres aléatoires (`rand` et `srand`) n'est pas spécifié dans la norme. Le comportement de ces fonctions n'est donc pas portable et non prédictible et pose des questions concernant la qualité et les performances. La fonction `rand` n'est pas pour le moment marquée comme dépréciée, il est simplement fortement recommandé de ne plus l'utiliser.

3 Les nouvelles fonctionnalités sur les fonctions

3.1 Relâchement des contraintes sur constexpr (N3652)

Une expression en C++ peut être évaluée lors de la compilation, ce qui permet d'être plus efficace lors de l'exécution. Ainsi, le simple code suivant ne sera pas compilé en une addition, mais sera directement remplacé par une constante (même si l'on ne comprend pas l'assembleur, on voit que la valeur 120 est directement dans le code généré) :

```
1    const int i = 1*2*3*4*5;
2    // sera compilé en (Clang 3.4.1) :
3    // movl $120, %eax
```

Le C++11 introduit le mot-clé `constexpr` pour créer des fonctions qui peuvent être exécutées à la compilation. Cela permet de déclarer des calculs comme des fonctions sans devoir en payer le prix à l'exécution. Cependant, ces fonctions étaient très limitées et n'autorisaient que des calculs simples (une seule expression à retourner principalement). Par exemple, pour calculer une factorielle, le code suivant n'est pas évalué à la compilation :

```
1    int factorial(int n)
2    {
3        return n <= 1 ? 1 : (n * factorial(n-1));
4    }
5
6    int main() {
7        const int i = factorial(5);
8    }
```

Le C++14 (N3652) permet de lever les limitations de `constexpr`, il est donc maintenant possible, dans une fonction `constexpr` :

- de déclarer des variables ;
- d'utiliser des expressions conditionnelles (`if`, `switch`) ;
- d'utiliser des boucles (`while`, `for`) ;
- de créer et modifier des objets ;
- etc.

Même si des restrictions importantes restent (les objets doivent être créés à l'intérieur de la fonction, il est possible d'appeler uniquement d'autres fonctions `constexpr`, etc.) leur

usage va pouvoir se développer (et oblige probablement le compilateur à disposer d'un interpréteur C++).

Autre exemple d'utilisation de `constexpr`, avec une boucle `for` et un test `if`, pour déterminer la valeur minimale d'une liste de valeurs (`initializer_list`) :

```
constexpr int min(std::initializer_list<int> xs) {
    int min = std::numeric_limits::max();
    for (int x: xs) {
        if (x < min) {
            min = x;
        }
    }
    return min;
}

constexpr int z = min({ 1, 3, -2, 4 });
```

3.2 Généralisation de l'inférence de type

3.2.1 Rappels sur l'inférence de types en C++

Le C++11 a introduit la déduction de types (inférence de types) sous deux formes : `auto` et `decltype()`. Cette fonctionnalité permet ainsi de laisser le compilateur choisir le type d'un élément en fonction de ce qui lui est assigné.

Le mot-clé `auto` permet de laisser faire automatiquement le choix. Par exemple :

```
int i = 0;
auto j = i + 2; // i est un int, 2 aussi donc j sera du type int.
```

Le mot-clé `decltype` permet de déduire le type d'une autre expression. Par exemple :

```
int i = 0;
double k = 2.0;
decltype(k) j = i + 2; // j sera du même type que k, donc double.
```

L'intérêt de `decltype` est ici limité, mais devient vite indispensable lorsque certaines variables utilisent `auto` (car seul le compilateur possède le type explicite) ou lors de l'utilisation de templates (puisqu'on ne connaît pas le type à l'avance).

Il est également possible d'utiliser l'inférence de type pour le retour d'une fonction. La syntaxe générale consiste à utiliser `auto` comme type de retour et à indiquer le type après la signature de la fonction avec `-> type` (en utilisant `decltype` ou non).

```
    // sans decltype
    auto add_int(int lhs, int rhs) -> int {
        return lhs + rhs;
    }

    // avec decltype
    template<typename LHS, typename RHS>
    auto add(LHS lhs, RHS rhs) -> decltype(lhs + rhs) {
        return lhs + rhs;
    }

    auto i = add(1, 1);   // int
    auto j = add(1, 1.0); // double
```

Dans le cas des fonctions lambda, il est possible d'omettre la déclaration de type avec `->` si le corps de la fonction ne contient qu'un seul `return` :

```
    [](int rhs, int lhs) -> int { return rhs + lhs; }; // ok
    [](int rhs, int lhs)        { return rhs + lhs; }; // ok
```

3.2.2 Utilisation de decltype(auto)

Une différence importante entre `auto` et `decltype` est que le premier ne conserve pas les qualificateurs (`const`, `&` et `&&`). Ainsi, si l'on écrit :

```
    {
        const int i { 12 };
        auto j = i;
        j += 34;
    }
    {
        const int i { 12 };
        decltype(i) j = i;
        j += 34; // error: read-only variable is not assignable
    }
```

De la même façon, avec des références :

```
    int&& foo();
    auto i = foo();             // int
    decltype(foo()) j = foo(); // int&&
```

La nouvelle expression `decltype(auto)` vient ainsi mixer les deux expressions : elle se place partout où un `auto` peut être utilisé pour une déduction de type, mais reprend le même comportement que `decltype()`.

3.2.3 Uniformisation des syntaxes

Le premier changement apporté par le C++14 (N3638) est d'uniformiser les syntaxes entre les fonctions lambda et non lambda pour le retour de type. Il est donc possible d'omettre `-> type` dans les fonctions non lambda, dans les mêmes conditions que pour les fonctions lambda.

```
1   // C++11
2   template<typename LHS, typename RHS>
3   auto add(LHS lhs, RHS rhs) -> decltype(lhs + rhs) {
4       return lhs + rhs;
5   }
6
7   // C++14
8   template<typename LHS, typename RHS>
9   auto add(LHS lhs, RHS rhs) {
10      return lhs + rhs;
11  }
```

3.2.4 Simplification des règles d'utilisation de ->

Le second changement apporté par le C++14 est qu'il est possible d'omettre `-> type` dans une fonction qui possède plusieurs `return`, à partir du moment où tous les `return` renvoient le même type.

Par exemple, pour calculer une factorielle :

```
1   auto fact(int n) // pas besoin de ->
2   {
3       int ret = 1;
4       for (int i = 1; i <= n; i++)
5           ret *= i;
6       return ret;
7   }
```

Dans le cas de méthodes récursives, ce genre de construction n'est possible que si les appels récursifs apparaissent après un premier retour de fonction. Ainsi ce code est correct :

```
1   auto fact(int n)
2   {
3       if (n <= 1)
4           return 1; // Ok : Un retour de fonction avant
5                     // l'appel récursif
6       else
7           return n * fact(n-1);
8   }
```

Tandis que celui-ci provoquera une erreur de compilation :

```
auto fact(int n)
{
    if (n > 1)
        return n * fact(n-1); // KO : Appel récursif avant d'avoir
                              // vu un retour de fonction
    else
        return 1;
}
```

3.2.5 les fonctions lambda génériques (N3649)

Le C++11 avait introduit les fonctions anonymes (lambda) :

```
[](int x, int y) -> int { return x + y; }
```

La norme spécifiait de quelle manière ces expressions devaient être transformées, en utilisant des classes classiques et un opérateur d'appel `operator()` sans templates. Cette précision interdisait de fait la déduction de types et tous les types des paramètres de fonctions devaient donc être spécifiés. Cette restriction a été levée (N3649) et il est donc maintenant possible de faire des fonctions anonymes polymorphiques (entre autres grâce à l'introduction de la déduction de type de retour présenté ci-dessus) :

```
auto add = [](auto a, auto b){return a + b;}
```

Remarque : le TS (Technical Specification) "Concept" élargit l'utilisation de `auto` dans les paramètres de fonctions autres que les lambda. Cette fonctionnalité est implémentée dans GCC 4.9 en utilisant la directive `-std=c++14`. Il est donc possible d'écrire :

```
auto add(auto i, auto j) {
    return i+j;
}
```

3.3 Expression de captures dans les lambda (N3648)

3.3.1 Limitations des captures dans les lambda

Les captures dans les fonctions lambda permettent de passer des variables dans les fonctions lors de la création. Ainsi, dans le code suivant, la valeur de la variable `i` est "capturée" une seule fois, lors de la création de la fonction, tandis que la valeur de la variable `j` est modifiée à chaque appel de la fonction.

```
int i = 5;
auto f = [i](int j) { return i * j; };
f(10);  // 50
f(100); // 500
```

Cependant, la capture présente des limitations. Par exemple, il n'est pas possible de passer une valeur directement dans la capture, il faut créer une variable et passer cette variable.

```
auto f = [i=5](int j) { return i * j; };
```

De la même façon, il n'est pas possible d'utiliser le déplacement (*move semantic*) dans les captures. Prenons un exemple concret : supposons que vous avez une ressource gérée par un pointeur intelligent `unique_ptr` et que vous souhaitez utiliser cette ressource dans un nouveau thread. Le code suivant ne fonctionne pas :

```
unique_ptr<Ressource> r = create_ressource();
auto use_ressource = [r](){ do_something(r); };
std::thread t(use_ressource);
```

Ce code produit l'erreur suivante, puisque `r` n'est pas copiable :

```
error: call to implicitly-deleted copy constructor of
'std::unique_ptr<int>'
    auto use_ressource = [r](){ do_something(r); };
```

3.3.2 Les approches possibles

Plusieurs solutions sont possibles en C++11. Par exemple, il est possible de passer la ressource par référence :

```
unique_ptr<Ressource> r = create_ressource();
auto use_ressource = [&r](){ do_something(r); };
std::thread t(use_ressource);
```

Cependant, cette approche pose un problème de gestion de la durée de vie de la ressource. La référence créée `&r` est un observateur de la ressource, mais pas un propriétaire (*ownerwhip*). Le premier thread reste le propriétaire, il y a un risque de destruction de la ressource alors qu'elle est encore utilisée par le thread :

```
{
    unique_ptr<Ressource> r = create_ressource();
    auto use_ressource = [&r](){ do_something(r); };
    std::thread t(use_ressource);
} // problème : r est détruit, mais la ressource est
  // encore utilisée dans le thread
```

Comme le premier thread est propriétaire de la ressource, c'est à lui de vérifier que cette ressource n'est plus utilisée avant de la supprimer. Par exemple, en attendant la fin de l'exécution du thread :

```
{
    unique_ptr<Ressource> r = create_ressource();
    auto use_ressource = [&r](){ do_something(r); };
    std::thread t(use_ressource);
    t.joint(); // ok... mais pas performant
}
```

On perd ici l'intérêt d'utiliser un thread, puisque l'on est obligé d'attendre qu'il a fini. Il faut donc obligatoirement donner la propriété de la ressource au thread, pour que le premier thread n'ait à s'en occuper. En C++11, la solution est de partager la responsabilité en utilisant `shared_ptr` au lieu de `unique_ptr` :

```
{
    shared_ptr<Ressource> r = create_ressource();
    auto use_ressource = [r](){ do_something(r); };
    std::thread t(use_ressource);
}
```

Il est également possible d'utiliser une classe (foncteur), qui va prendre la responsabilité du `unique_ptr`, mais c'est plus lourd à mettre en place. Dans tous les cas, il n'est pas possible de simplement déplacer la ressource dans le nouveau thread.

3.3.3 Les nouvelles captures du C++14

Le C++14 (N3648) permet d'utiliser des références sur des rvalues (*RValues references*) et de définir n'importe quelle variable propre à la fonction anonyme.

Il est donc possible de créer une variable interne à la fonction lambda :

```
// création d'une valeur
auto f = [i = 5](int j) { return i * j; };

// création d'un weak_ptr
shared_ptr<Ressource> r = create_ressource();
auto use_ressource = [wr = weak_ptr<Ressource>(r)]()
    { do_something(wr); };
std::thread t(use_ressource);
```

Il est également possible d'utiliser le déplacement :

```
unique_ptr<Ressource> r = create_ressource();
auto use_ressource = [ur = std::move(r)](){ do_something(ur); };
std::thread t(use_ressource);
```

4 Modifications sur la bibliothèque standard

4.1 De nouveaux suffixes pour les littéraux (N3531)

Une littérale est une valeur constante écrite directement dans le code. Chaque littérale possède son propre type, défini par la norme. Cela est particulièrement important avec l'utilisation de `auto`, la déduction de type étant faite en utilisant la littérale :

```
1    auto i = 123;              // int
2    auto d = 12.34;            // double
3    auto s = "hello, world";   // const char*
4    auto l = { 1, 2, 3, 4 };   // std::initializer_list<int>
```

Lorsqu'une littérale est affectée à une variable d'un type différent, une conversion implicite est réalisée. L'utilisation des crochets permet de générer un message d'avertissement lorsqu'il y a un risque de perte d'information :

```
1    unsigned i { -1 };
```

Produit l'erreur :

```
main.cpp:2:18: error: constant expression evaluates to -1 which
cannot be narrowed to type 'unsigned int' [-Wc++11-narrowing]
    unsigned i { -1 };
                 ^~
main.cpp:2:18: note: insert an explicit cast to silence this issue
    unsigned i { -1 };
                 ^~
                 static_cast<unsigned int>( )
```

Les suffixes de littéraux permettent de modifier le type d'une littérale :

```
1    auto i = 123u;    // unsigned int
2    auto j = 123l;    // long int
3    auto f = 12.34f;  // float
```

Il existe plusieurs dizaines de suffixes, voir la documentation pour les détails : Literals.

Si on souhaite créer une littérale d'un type différent, il faut convertir explicitement cette littérale, ce qui rend l'écriture un peu lourde :

```
auto i = std::size_t { 123 };
auto s = std::string { "hello, world" };
```

Le C++11 a ajouté la possibilité d'écrire ses propres suffixes littéraux, en utilisant l'opérateur `operator""` (User-defined literals). Il est ainsi possible de créer une littérale de n'importe quel type :

```
class my_class {
    int value {};
public:
    my_class(int i) : value{ i } {
    }
};

constexpr my_class operator"" _mc (int i) {
    return my_class { i };
}

auto x = 123_mc; // type my_class
```

Pour simplifier la création de littérales, le C++14 ajoute de nouveaux suffixes littéraux, pour quelques classes de la bibliothèque standard :

- le suffixe s pour `std::string` ;
- les suffixes i, if et il pour `std::complex` (pour créer un nombre complexe de valeur réelle nulle et de type respectivement `std::complex<double>`, `std::complex<float>` et `std::complex<long double>`) ;
- les suffixes h, min, s, ms, us et ns pour `std::chrono::duration` (pour respectivement des heures, des minutes, des secondes, des millisecondes, des microsecondes et des nanosecondes).

```
auto s = "hello, world"s;    // std::string
auto c = 123i;               // std::complex<double>
auto t = 14h + 27min + 34s; // std::chrono::duration
```

(Attention de bien distinguer le suffixe `s` pour les chaînes et pour les nombres.)

4.2 Make_unique

Alors que le besoin de `std::make_shared` s'est fait sentir dès l'ajout de `std::shared_ptr` (voir le Guru Of The Week #89, en particulier les deux images dans l'article), `std::make_unique` n'a pas été intégré tout de suite dans les nouvelles normes du C++.

La méthode habituelle pour créer un objet managé par `std::unique_ptr` est d'utiliser `new` lors de la création ou dans la fonction `reset` :

```
1    std::unique_ptr<A> p(new A);
2    p.reset(new A);
```

Ce code ne présente pas de risque de fuite mémoire, puisque l'objet créé avec `new` est tout de suite pris en charge par `unique_ptr`.

Il existe cependant des cas où l'utilisation de `new` avec `unique_ptr` présente des risques de fuites mémoire : lorsque vous avez plusieurs créations d'objets dans une même expression. Par exemple :

```
1    void foo(std::unique_ptr<A> a, std::unique_ptr<A> b);
2    foo(new A, new A);
```

Dans ce cas, si le second `new` lance une exception, le premier objet créé n'est pas encore managé et il ne sera pas supprimé.

L'ajout de `make_unique` en C++14 permet d'éviter ce risque.

```
1    void foo(std::unique_ptr<A> a, std::unique_ptr<A> b);
2    foo(std::make_unique<A>(), std::make_unique<A>());
```

L'autre intérêt est purement syntaxique : avec `make_unique` et `make_shared`, vous pouvez complètement interdire l'utilisation de `new` et `delete` dans un code C++14.

Remarque : en C++11, il est préférable de quand même créer la fonction `make_unique` et d'utiliser une compilation condition pour choisir entre le `make_unique` que vous aurez créé et le `std : :make_unique` du C++14.

5 Les autres apports du C++14

Le brouillon propose tout un tas d'autres corrections qui sont principalement là pour clarifier certaines situations et n'ont donc pas été citées ici. Certaines modifications, proposées à l'origine, ont aussi été refusées ou repoussées à la prochaine norme. Nous avons donc peut être oublié certaines de ces modifications, n'hésitez pas à nous la signaler en commentaire si c'est le cas.

- N3323 - A Proposal to Tweak Certain C++ Contextual Conversions, v3 ;
- N3653 - Member initializers and aggregates ;
- N3664 - Clarifying Memory Allocation ;
- N3778 - C++ Sized Deallocation ;
- N3887 - Consistent Metafunction Aliases ;
- N3891 - A proposal to rename shared_mutex to shared_timed_mutex ;
- N3910 - What can signal handlers do ? ;
- N3927 - Definition of Lock-Free.

6 Et maintenant ?

Le comité de normalisation va maintenant faire quelques modifications mineures (principalement typographiques) et envoyer le document à l'ISO pour une publication avant la fin de l'année. Dès maintenant, le comité continue son travail et commence la prochaine mise à jour. En effet, il est prévu de voir apparaître la prochaine évolution du C++ en 2017 si aucun retard n'est pris. Contrairement au C++14, et dans la lignée du C++11, cette prochaine mise à jour est prévue pour être une évolution majeure.

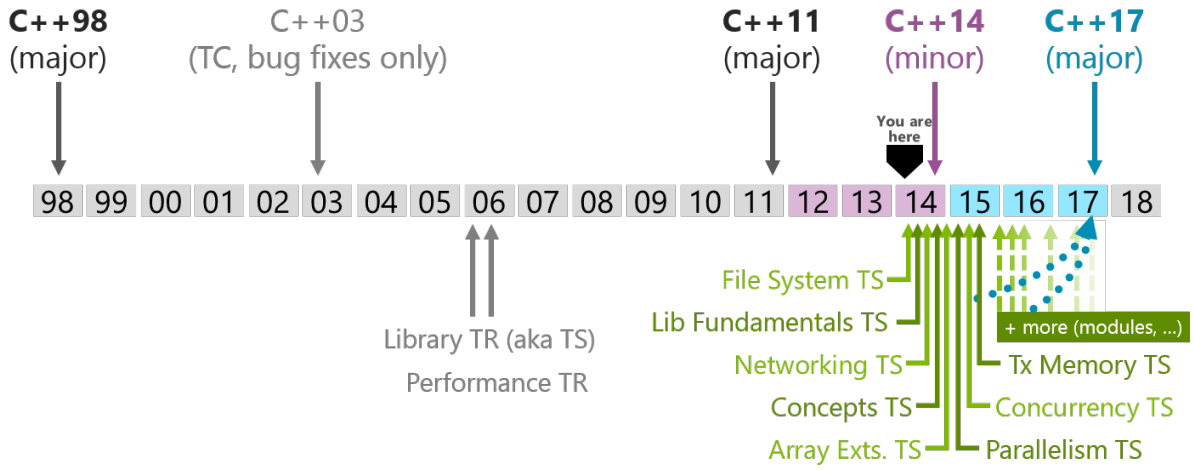

FIGURE 6.1: (Source : isocpp.org)

La prochaine norme contiendra en particulier plusieurs modifications qui étaient prévues pour le C++14 et qui ont dû être reportées. Elle contiendra également de nombreuses modifications importantes, qui seront disponibles avant 2017 sous forme de TS (*Technical Specification*) :

- File System : gestion de fichiers et dossiers, similaire à Boost.Filesystem ;
- Library Fundamentals : ensemble d'extensions pour la bibliothèque standard, par exemple optional, any, string_view ;
- Networking : support du réseau ;
- Concepts : validation des types template ;
- Array : extensions concernant les tableaux, en particulier `dynarray` ;
- Parallelism : accès aux fonctionnalités de parallélisation des processeurs ;
- Concurrency TS : programmation concurrente ;
- Transactional Memory : mémoire transactionnelle.

En complément, des groupes d'étude étudient les évolutions futures du langage et la bibliothèque standard. On peut citer par exemple :

- Modules (SG2) : amélioration du système d'en-têtes ;

– Reflection (SG7) : utilisation de la réflexivité ;
– Undefined and Unspecified Behavior (SG12).

En attendant, LLVM et GCC proposent déjà des implémentations quasi complètes du C++14, une première ! Il avait fallu attendre 2 ans pour avoir la première implémentation complète du C++11 et 5 ans pour le C++98 ! Le C++ semble retrouver une nouvelle dynamique et continue d'évoluer.

6.1 Support du C++14

– Statut du support C++14 dans Clang
– Statut du support C++14 dans libc++
– Statut du support C++14 dans GCC
– Statut du support C++14 dans libstdc++
– Statut du support C++11/14 dans VS
– Statut du support C++11/14 de la STL dans VS14 CTP1
– Comparaison du support des différents compilateurs et bibliothèques

7 Sources

- Looking at C++14 (Metting C++)
- We have C++14 ! (isocpp.org)
- C++14 (Wikipédia)
- Clang 3.4 and C++14
- FAQ C++ (isocpp.org) : Overview, Language et Library.

7.1 Pour aller plus loin

- C++Now 2014 (BoostCon, YouTube)
- GoingNative 2013 (Channel 9)
- Nico Josuttis - C++14 (NDC 2014)