[Aller plus loin] Le perfect forwarding

lvalue/rvalue : oui, on peut effectivement simplifier en “avec une adresse memoire” (les variables) vs “le reste” (les valeurs temporaires, en gros, les expressions et les literales)

lvalues references : tu sembles plus decrire un pointeur, mais c'est d'idee. (En interne, sur un appel de fonction, une lvalue ref sera un pointeur cache. Hors fonction, ca sera plus un alias de variable. Dans tous les cas, c'est un indirection sur une lvalue).

rvalue references : comme il n'y a pas d'adresse memoire pour une rvalue, non. Mais en gros, c'est pareil que pour une lvalue ref : c'est une indirection sur un rvalue. Peut importe comment le compilateur gere cela en interne.

universal reference : reference sur une rvalue ou une lvalue. C'est un template (ou de la deduction de type), donc lors de l'instanciation du template, la substitution du type definira si c'est une lvalue ref ou une rvalue ref. Donc une universal ref s'ecrit toujours :

```
template<class T>
void f(T&& uref);

// ou

void g(auto&& uref)
```

(donc && avec devant pas un vrai type, mais un template/auto)

Pourquoi plusieurs types de references : pour la surcharge.

```
void f(int &);  // #1
void f(int &&); // #2
```

```
int i;
f(i); // i est une lvalue -> appel d'une lvalue ref.
      // #1 est appele


f(123); // 123 est une rvalue -> appel d'une rvalue ref.
        // #2 est appele
```

Rien de bien complique : chaque type de reference fonctionne avec un type de value specifique.

On en arrive au forwarding. C'est simplement le fait de passer une variable dans plusieurs fonctions.

```
void f(int);

void g(int i) { f(i); }  // g forward i dans f

f(123);  // appel direct de f
g(123);  // appel indirect de f
```

Maintenant, on ajoute les references...

```
#include <iostream>

// #f1
void f(int &) {
    std::cout << "#f1" << std::endl;
}

// #f2
void f(int &&) {
    std::cout << "#f2" << std::endl;
}

// #g1
void g(int & i) {
    std::cout << "#g1 "; f(i);
}
```

```
// #g2
void g(int && i) {
    std::cout << "#g2 "; f(i);
}

int main() {
    int i {};

    f(i);   // appel direct de #f1 ?
    g(i);   // appel indirect de #f1 ?

    f(123); // appel direct de #f2 ?
    g(123); // appel indirect de #f2 ?
}
```

Maintenant, le piege...

Si j'ai ecrit un code complet, c'est pour le tester reellement. Et cela affiche :

```
#f1
#g1 #f1
#f2
#g2 #f1
```

Et le probleme est la : la derniere fonction appelle #f1 et pas #f2. L'appel indirect n'appelle pas la meme fonction que l'appel direct.

Le perfect forwarding est simplement lorsque les appels de fonctions appellent le meme type de value et references. (Donc, si le perfect forwarding fonctionnait dans ce code, la derniere ligne devrait etre #f2).

(Le “perfect forwarding” signifie “passage/transmission parfaite”, c'est a dire quand le type de value est respectee lors des appels de fonctions, ie qu'une lvalue ne soit pas convertie en rvalue et reciproquement)

Maintenant, l'explication du piege :

```
void f(int         i);
void f(int &       i);
void f(int const&  i);
```

```
void f(int &&       i);
void f(auto &&      i);
```

Dans ce code, quelle sont les parametres “i” qui sont des lvalue et ceux qui sont des rvalues ?

…

suspense…

…

Tous les parametres “i” sont des lvalue ! Dans tous les cas, ce sont des variables en memoire, qui ont une adresse, donc des lvalue.

Par contre, ce sont des lvalue (des variables) de differents types. Il faut distinguer le type (au sens du C++) et le “type de value” (ie lvalue ou rvalue). Par exemple, “int i”, i est une lvalue de type “int”, dans “int & i”, i est une lvalue de type “lvalue reference sur int”, et dans “int && i”, i est une lvalue de type “rvalue reference sur int”.

Dans la fonction #g2 precedent, i est une lvalue de type “rvalue reference sur int”. C'est donc #f1 qui est appele par #g2 (puisque #f1 contient une lvalue reference et accepte donc une lvalue comme argument).

Note : il est possible de corriger #g2 avec std::move. Mais on doit ecrire 2 fonctions g differentes. Si f avait 2 parametres, il faudrait ecrire 4 fonctions g, et si f avait 3 parametres, il faudrait ecrire 8 fonctions g… cela devient vite complique.

Et maintenant, le code corrige avec std::forward :

```
#include <iostream>

// #f1
void f(int &) {
    std::cout << "#f1" << std::endl;
}

// #f2
```

```
void f(int &&) {
    std::cout << "#f2" << std::endl;
}

// #g
template<class T>
void g(T && i) {
    std::cout << "#g "; f(std::forward<T>(i));
}

int main() {
    int i {};

    f(i);   // appel direct de #f1 ?
    g(i);   // appel indirect de #f1 ?

    f(123); // appel direct de #f2 ?
    g(123); // appel indirect de #f2 ?
}
```

affiche :

```
#f1
#g #f1
#f2
#g #f2
```

Ok, on a du perfect forwarding ici, f2 est correctement appelee.

Et quand on ecrit une fonction “make” comme smartPointerMaker, c'est bien du perfect forwarding que l'on veut : on veut que la construction de l'objet via smartPointerMaker soit strictement equivalent a l'appel direct de new.

Chapitre précédent	Sommaire principal	Chapitre suivant