Zeste de savoir

La déduction de type en C++

6 mars 2021

Table des matières

1. Les règles de déduction des templates . . . 2
   1.1. Résumé . . . 7
2. Passons en mode auto . . . 7
   2.1. Quelques exemples . . . 8
   2.2. Des petites exceptions . . . 10
   2.3. Résumé . . . 15
3. Mais qu'est-ce que decltype? . . . 16
   3.1. Exemples d'utilisation . . . 16
   3.2. Le petit plus de C++14 . . . 18
   3.3. La règle exacte de `decltype` . . . 19
   3.4. Résumé . . . 23
4. Connaître et afficher le type exact . . . 23
   4.1. Vérifier de façon statique . . . 23
   4.2. Vérifier de façon dynamique . . . 25
5. Quand les utiliser? . . . 27

L'inférence de type est la capacité qu'à un compilateur à déduire le type d'une expression sans que celui-ci ne soit explicitement écrit dans le code source. C'est une fonctionnalité très intéressante puisqu'elle permet, quand elle est bien utilisée, de **rendre le code plus lisible** et d'**éviter de la redondance**.

L'inférence de type existe depuis bien longtemps en C++. Eh oui! les templates, ça te dit quelque chose? Alors, pourquoi s'en soucier et en faire un tutoriel? C'est que depuis 2011, puis en 2014 et encore en 2017, **les choses ont changé**. Le mots-clef `decltype` est apparu et `auto` a pris un nouveau sens; de plus, les lambdas, les fonctions avec le type de retour déduit automatiquement, etc, sont autant de nouveaux cas où l'on va rencontrer la déduction de type et apportent leurs lots de petites subtilités (sinon, ce n'est pas amusant).

Alors examinons un peu plus en profondeur comment ça marche, pour que le C++ moderne ne te cache aucune surprise.

i Avant tout

Prérequis
Connaître les bases du C++ , dont les templates.
Connaître les notions de lvalues et de rvalues.
Objectifs
Comprendre comment fonctionne l'inférence de type, dans toutes ses facettes.
Remarques
Les codes montrés en exemple sont inspirés de ceux écrits par Scott Meyers dans son livre Effective Modern C++ , comme autorisé par l'auteur, tout comme certaines explications.

1. Les règles de déduction des templates

Les templates sont un mécanisme bien connu des développeurs C++ puisqu'ils sont présents absolument partout dès que l'on a besoin de généricité. La bibliothèque standard en regorge. Le cas le plus simple se présente ainsi.

```cpp
// Déclaration
template <typename T>
void f(ParamType);  // Peut être différent de T, avec par exemple const ou &.

// Appel
f(expression);
```

Notre objectif est de trouver `T` pour que `ParamType` (appelé `P` dans la norme) et le type de `expression` (appelé `A` dans la norme) soient les mêmes. Heureusement pour nous, il existe une règle très simple.

Si `ParamType` contient une référence ou un pointeur, celui-ci est ignoré pour la déduction de `T`.

> If P is a reference type, the type referred to by P is used for type deduction.
>
> *N3690 §14.8.2.1.2*

Illustration par du code.

```cpp
template <typename T>
void function(T & parameter);

int i = 42;   // Ici, i est de type int.
function(i);  // Donc ParamType est de type int& et T de type int.

const int j = i;  // Ici, j est de type const int.
function(j);      // Donc ParamType est de type const int& et T de type const int.

const int & k = i;  // Ici, k est de type const int&.
function(k);        // Donc ParamType est de type const int& et T de type const int.

// ------------------------------------ //

template <typename T>
void another(T * parameter);
```

```
17
18 int a = 42;    // Ici, a est de type int.
19 another(&a);   // Donc ParamType est de type int* et T de type int.
20
21 const int * ptr = &a;   // Ici, ptr est de type const int*.
22 another(ptr);          // Donc ParamType est de type const int* et
       T de type const int.
```

1.0.1. Ignorons donc `const`

La règle n'est pas encore complète. En effet, il est fréquent de voir des fonctions prenant des arguments constants et, plus rare mais possible, des fonctions par recopie. Que se passe-t-il dans ce cas?

i

Si `ParamType` contient `const`, ou bien si `ParamType` ne contient ni référence (que ce soit `ParamType&` ou `ParamType&&`) ni pointeur, le qualificateur `const` est également ignoré pour la déduction de `T`.

> If P is not a reference type, and if A is a cv-qualified type, the top level cv-qualifiers of A's type are ignored for type deduction.
>
> *N3690 §14.8.2.1.2*

> If P is a cv-qualified type, the top level cv-qualifiers of P's type are ignored for type deduction.
>
> *N3690 §14.8.2.1.3*

```
1  template <typename T>
2  void constness(const T & parameter); // parameter est toujours
       constant.
3
4  int x = 42;     // Ici, x est de type int.
5  function(x);    // Donc ParamType est de type const int& et T de
       type int.
6
7  const int y = x;  // Ici, y est de type const int.
8  function(y);      // Donc ParamType est de type const int& et T de
       type int.
9
10 const int & z = y;  // Ici, z est de type const int&.
11 function(z);        // Donc ParamType est de type const int& et T
       de type int.
12
13 // ------------------------------------ //
```

```
14
15 template <typename T>
16 void function(T parameter);  // Passage par recopie.
17
18 int x = 0;     // Ici, x est de type int.
19 foo(x);        // Selon la règle, ParamType et T sont de type int.
20
21 const int y = x;  // Ici, y est de type const int.
22 foo(y);           // Selon la règle, ParamType et T sont de type
       int.
23
24 const int & z = x;  // Ici, z est de type const int&.
25 foo(z);             // Selon la règle, ParamType et T sont de type
       int.
```

Quand on y réfléchit, cette règle est parfaitement logique. Pour rappel, nous cherchons un `T` qui rend identiques les types `A` et `P`. Ce dernier est déjà défini comme `const`, donc `T` n'a pas besoin de l'être.

Quant à la deuxième partie, elle aussi est logique, parce que si je manipule une copie de l'objet, je suis en droit de le modifier à ma guise, que m'importe si l'original est immutable.

1.0.2. Les références universelles

Une petite subtilité, qui nous vient de ces fameuses références universelles `T&&` apparues avec C++11 (aussi appelées, dans la norme, *forwarding references*), et la règle sera complète.

i

Si `ParamType` est de la forme `T&&` (et **uniquement** de cette forme) et que **`expression` est une lvalue**, alors `ParamType` et `T` sont déduits tous les deux comme `T&`.

> A forwarding reference is an rvalue reference to a cv-unqualified template parameter. If P is a forwarding reference and the argument is an lvalue, the type "lvalue reference to A" is used in place of A for type deduction.
>
> *N4164*

```
1 template <typename T>
2 void function(T && parameter);
3
4 int i = 42;   // Ici, i est une lvalue de type int.
5 function(i);  // Selon la règle, T et ParamType sont déduits comme
      int&.
6
7 int & j = i;  // Ici, j est une lvalue de type int&.
```

```
8  function(j);  // Selon la règle, T et ParamType sont déduits comme
       int&.
9
10 const int & k = i;   // Ici, k est une lvalue de type const int&.
11 function(k);         // Selon la règle, T et ParamType sont déduits
       comme const int&.
12
13 function(0);    // Ici, 0 est une rvalue de type int, donc T est
       déduit comme int et ParamType comme int&&.
```

Pour bien comprendre, il faut déjà savoir qu'une référence universelle se comporte soit comme une référence sur une lvalue, soit comme une référence sur un rvalue. Sachant ceci, reprenons le code ensemble.

```
1 int i = 42;
2 function(i);
```

Premier cas, `i` est une lvalue de type `int`, donc notre référence universelle va se comporter comme une référence sur une lvalue, soit `int&`. Notre fonction attend donc un type `int& &&`. La lvalue prévaut sur la rvalue, donc `&&` est enlevé et tant `ParamType` que `T` sont déduits comme `int&`.

```
1 int & j = i;
2 function(j);
```

Deuxième cas, `j` est une référence sur une lvalue de type `int`, soit `int&`. Notre référence universelle va se comporter, une fois de plus, comme une référence sur une lvalue. Notre fonction attend donc un type `int& &&`. La lvalue prévaut sur la rvalue, donc `&&` est enlevé et tant `ParamType` que `T` sont déduits comme `int&`.

```
1 const int & k = i;
2 function(k);
```

Troisième cas, `k` est une référence constante sur une de type `int`, soit `int const&`. Cette fois, notre fonction attend un type `int const& &&`. Comme précédemment, la lvalue prévaut sur la rvalue, donc `&&` est enlevé et tant `ParamType` que `T` sont déduits comme `int const&`.

```
1 function(0);
```

Quatrième et dernier cas, cette fois on passe une rvalue à la fonction, donc notre référence universelle se comporte comme une référence sur une rvalue. Notre fonction attend un `int &&`,

nous lui en passons un. La règle générale s'applique, on supprime toute référence du type `T` et celui-ci est déduit comme un simple `int`.

Avez-vous remarqué qu'une référence sur une lvalue prévaut **toujours** sur une référence sur une rvalue? C'est ce qu'on appelle la *reference collapsing rules* [1]. C'est pour cela que, quand nos fonctions attendent des `int& &&` ou des `int&& &`, le type déduit est toujours `int&`. Il n'y a que dans le cas où l'on passe un `int&&` à une fonction attendant un `int&&` en paramètre que la rvalue est conservée.

Là encore, point d'exception, une simple subtilité toute logique, dès lors qu'on connaît cette règle de *reference collapsing*.

1.0.3. Les tableaux C

Je sais, un bon programmeur C++ doit privilégier `std::array` ou `std::vector` à la place de cet héritage du C. L'univers n'étant pas parfait, il arrive de devoir travailler avec. Et puis ils vont illustrer les règles de déduction de façon utile et amusante.

Tout d'abord, il est important de savoir qu'**il est impossible de passer un tableau par recopie**. En effet, les règles héritées du C impliquent que, dans quasiment toutes les situations, un tableau est converti en un pointeur constant sur son premier élément. Les deux fonctions suivantes sont donc strictement **identiques**.

```
1 void first(int * parameter);
2 void second(int parameter[]);
```

Donc si l'on passe un tableau à une fonction qui prend des arguments par recopie, celui-ci sera quand même converti en pointeur.

```
1 template <typename T>
2 void function(T parameter);
3
4 const char str[] = "Zeste de Savoir";  // Ici, str est de type
      const char[15].
5 function(str);                          // Conversion en pointeur, T
      sera déduit comme un const char*.
```

> If P is not a reference type, and if A is an array type, the pointer type produced by the array-to-pointer standard conversion (4.2) is used in place of A for type deduction.
>
> *N3690 §14.8.2.1.2*

1. Règle détaillée ici plus en détails.

Et si on déclare une fonction prenant une référence et qu'on lui passe le tableau? Eh bien la règle dictée plus haut s'applique et T est déduit comment étant de même type que le tableau.

```
template <typename T>
void function(T & parameter);

const char str[] = "Zeste de Savoir";  // Ici, str est de type
    const char[15].
function(str);                         // Suivant la règle, T sera
    de type const char[15] lui aussi.
```

Profitant de la possibilité d'instancier implicitement un template, on peut donc écrire un code comme celui ci-dessous, qu'on trouve sur de nombreux forums et qui permet de récupérer la taille d'un tableau C.

```
template <typename T, std::size_t N>
// Parenthèses nécessaires pour avoir une référence sur un tableau
// et non un tableau de N références.
inline constexpr std::size_t array_size(T (&)[N]) noexcept
{
    return N;
}
```

Les tableaux C ne constituent absolument pas une exception. Cette section avait juste pour but de montrer et d'expliquer ce code que l'on voit souvent sur les forums, tout en donnant un exemple concret de déduction de type.

1.1. Résumé

Les règles de déduction des templates ne sont pas compliquées. Faisons un petit résumé pour bien les fixer dans l'esprit.

- Si `ParamType` contient une référence ou un pointeur, celui-ci est ignoré pour la déduction de `T`.
- Si `ParamType` contient `const`, ou bien si `ParamType` ne contient ni référence (que ce soit `ParamType&` ou `ParamType&&`) ni pointeur, le qualificateur `const` est également ignoré pour la déduction de `T`.
- Si `ParamType` est une référence universelle et si l'expression passée est **une lvalue**, alors `ParamType` et `T` sont de type `T&`.

2. Passons en mode auto

Jusque là, nous n'avons pas vraiment abordé les nouveautés du C++ moderne. Il est temps d'y remédier en faisant connaissance avec `auto`, mot-clef qui nous autorise à faire de l'inférence de

type à souhait.

J'en vois déjà qui font la grimace en pensant à tout un tas de nouvelles règles compliquées qu'il va falloir digérer. Eh bien j'ai une bonne nouvelle pour vous tous.

Les règles de déduction utilisées par `auto` sont les mêmes que celles des templates.

En effet, `auto` se comporte comme notre `T`, si fréquent dans les templates. Ainsi, si l'expression contient une référence `&` ou `&&`, elle est ignorée; de même, `const` est ignoré tout aussi sauvagement.

2.1. Quelques exemples

```
1  int i = 42;
2
3  auto x = i;
4
5  // Correspond à ceci.
6  template <typename T>
7  void function(T parameter);
```

Voici le cas le plus simple. L'utilisation de `auto` entraîne la déduction du type de `x` comme étant `int`, ce qui est parfaitement logique, puisque c'est le type de `i`.

```
1  int i = 42;
2  int & j = i;
3
4  auto & x = j;
5
6  // Correspond à ceci.
7  template <typename T>
8  void function(T & parameter);
```

Notre variable `j` est une référence sur une lvalue de type `int`, donc est de type `int&`. Comme `auto` déduit `int` comme type, on ajoute manuellement une référence `&` pour que les types de `x` et de `j` correspondent. Oui, exactement comme pour la correspondance qu'on veut avoir dans le cas des templates.

```
1  const int i = 42;
2
3  auto & x = i;
4
```

```
// Correspond toujours à ceci.
template <typename T>
void function(T & parameter);
```

Allez, essaye de deviner le type déduit par `auto`. Si tu réponds `const int`, c'est que tu as bien compris les règles des templates. En effet, `i` est de type `const int`. Donc la référence sur une constante se doit d'être constante elle aussi, d'où le type déduit `const int`.

```
const int i = 42;

const auto & x = i;

// Correspond à ceci.
template <typename T>
void function(const T & parameter);
```

Là encore, rien d'étonnant. Nous obtenons `int` en type déduit, puisque nous avons précisé nous-mêmes que nous voulions une référence constante. Chacun est libre d'avoir son avis, mais je préfère justement cette forme plus explicite que le code précédent.

```
int i = 42;

auto * x = &i;

// Correspond à ceci.
template <typename T>
void function(T * parameter);
```

Allez, des pointeurs cette fois. Pour ne rien changer, `auto` déduit `x` comme étant un `int`, ce qui est exactement le même comportement que les templates.

2.1.1. Les tableaux C

```
const char site[] = "Zeste de Savoir";

auto pointer = site;
auto & array = site;
```

Juste pour réviser les tableaux en C. Comme vu dans la partie précédente, notre tableau est converti en un pointeur de type `const char*`, d'où `pointer` qui porte bien son nom. Mais en ajoutant une référence `&`, `array` est déduit comme étant une référence sur un tableau de `char` et donc déduit comme `const char (&)[15]`. Encore une fois, exactement comme les templates.

2.1.2. Les références universelles

```
1  int i = 42;
2  auto && x = i;
3
4  const int j = 0;
5  auto && y = j;
6
7  auto && z = 27;
8
9  // Correspond à ceci.
10 template <typename T>
11 void function(T && parameter);
```

Toujours rien de neuf sous le soleil, la loi des templates règne en maître. Dans notre code, `i` est une lvalue de type `int`. La *reference collapsing rule* s'applique, `&&` devient `&` et `x` est déduit comme `int&`, soit une référence sur une lvalue. De même, `y` est déduit comme `const int&`, car `j` est une lvalue de type `const int`.

Enfin, dans le cas de `z`, `27` est une rvalue de type `int`, donc `auto` déduit tout simplement le type comme étant `int` et `z` devient une référence sur une rvalue, soit `int&&`.

2.2. Des petites exceptions

Eh oui, petit lecteur innocent, **je ne t'ai pas tout dit**. Tu n'étais pas prêt à affronter la vérité toute nue qu'elle était.

i Il y a des cas où `auto` ne se comporte pas comme les templates.

2.2.1. `std::initializer_list`

Depuis 2011, parmi les ajouts au standard, on peut noter l'apparition de deux nouvelles façons d'initialiser nos variables: avec les accolades `{}`.

```
1 // Les formes classiques.
2 int x1 = 27;
3 int x2(27);
4 // Les deux nouvelles formes.
5 int x3 = {27};
6 int x4 {27};
```

Sauf que si l'on remplace `int` par `auto`, le résultat ↗ est surprenant puisque les deux dernières formes ne sont pas déduites comme étant des `int`, **mais comme des `std::initializer_list<int>` contenant un unique élément**.

```cpp
#include <iostream>
#include <utility>

void fun(int)
{
    std::cout << "Hey, I'm a function taking a int.\n";
}

void fun(const std::initializer_list<int> &)
{
    std::cout <<
        "Hey, I'm a function taking a std::initializer_list<int>.\n";
}

int main()
{
    auto x1 = 0;
    auto x2 (0);
    auto x3 {0};
    auto x4 = {0};

    fun(x1);  // Hey, I'm a function taking a int.
    fun(x2);  // Hey, I'm a function taking a int.
    fun(x3);  // Hey, I'm a function taking a
        std::initializer_list<int>.
    fun(x4);  // Hey, I'm a function taking a
        std::initializer_list<int>.

    return 0;
}
```

C'est là où réside la différence entre `auto` et les templates. Alors que l'utilisation d'accolades avec `auto` entraîne la déduction d'un `std::initializer_list<...>`, le code suivant avec les templates ne fonctionne tout simplement pas.

```cpp
#include <utility>

template <typename T>
void with_templates(const T &)
{

}

```

```cpp
template <typename T>
void with_initializer_list(const std::initializer_list<T> &)
{

}

int main()
{
    // Ici, auto entraîne la déduction d'un
        std::initializer_list<int> contenant 1, 2 et 3.
    // Donc auto diffère bien des templates sur ce point.
    auto x = {1, 2, 3};

    /* Clang
            Error: no matching function for call to
    'with_templates'.
            Note: candidate template ignored: couldn't infer
    template argument 'T'.

       GCC
            Error: no matching function for call to
    'with_templates(<brace-enclosed initializer list>)'.
            Note:   template argument deduction/substitution
    failed: couldn't deduce template parameter 'T'.
    */
    with_templates({1, 2, 3});

    // Par contre, aucun soucis ici.
    with_initializer_list({1, 2, 3});

    return 0;
}
```

La norme explique que, si le type déduit une fois les références et autres `const` enlevés est un `std::initializer<T>`, alors on opère une itération sur tous les éléments pour déduire `T`. Dans le cas contraire, le compilateur râle et plante.

> If removing references and cv-qualifiers from P gives std::initializer_list<P'> for some P' and the argument is an initializer list (8.5.4), then deduction is performed instead for each element of the initializer list, taking P' as a function template parameter type and the initializer element as its argument. Otherwise, an initializer list argument causes the parameter to be considered a non-deduced context (14.8.2.5).
>
> *N3690 §14.8.2.1.1*

Enfin, il reste quand même une subtilité dans la subtilité (une méta-subtilité pourrions-nous dire). La norme de 2017 change un peu les choses. Désormais, en C++17, dans le code suivant, la troisième forme d'initialisation donnera un `int` en lieu et place d'un `std::initializer_list<int>`. Si le nombre d'élément est supérieur à un, la compilation échouera.

```cpp
#include <iostream>
#include <utility>

int main()
{
    // C++11 / C++14 : x1 est déduit comme
        std::initializer_list<int> contenant un seul élément.
    // C++17 : x1 est déduit comme un int valant zéro.
    auto x1 {0};

    auto x2 = {0};  // Comme avant, x2 est déduit comme
        std::initializer_list<int>.

    auto x3 = {0, 1, 2}; // Comme avant, x3 est un
        std::initializer_list<int> de 3 éléments.

    // C++11 / C++14 : comme avant, std::initializer_list<int> de 3
        éléments.
    // C++17 : ne compilera pas car > 1 élement.
    auto x4 {0, 1, 2};
    // GCC : error : direct-list-initialization of 'auto' requires
        exactly one element.
    // Clang : error: initializer for variable 'x4' with type
        'auto' contains multiple expressions.

    return 0;
}
```

2.2.2. Les fonctions avec *trailing type*

Derrière ce nom se cache une nouvelle façon d'écrire des fonctions, identique à l'écriture des lambdas, avec l'utilisation de `auto`, comme suit.

```cpp
auto sum(int a, int b) -> int
{
    return a + b;
}
```

Avec C++11, c'est très simple, puisque ici **`auto` n'effectue aucune déduction et n'est là que pour la syntaxe**. Le type de retour de la fonction est, en effet, explicitement marqué en fin de ligne. Mais si je parle de ça, c'est que, tu t'en doutes, les choses ont changé.

Arrive 2014. Nouvelle version mineure de C++, avec ses ajouts. L'un d'entre eux va nous intéresser tout particulièrement: l'uniformisation des syntaxes des fonctions par rapport aux lambdas, avec la possibilité d'omettre `-> type`. Qu'est-ce qui change, concrètement?

Contrairement à C++11 où `auto` n'est là que pour la syntaxe, en C++14, une fonction avec `auto` comme type de retour fait de l'inférence de type **en utilisant les règles des templates**, une fois de plus. Ainsi, la fonction suivante n'est tout simplement pas valide.

```
1 // GCC : error: returning initializer list.
2 // Clang : error: cannot deduce return type from initializer list.
3 auto create_initialisation_list()
4 {
5     return {1, 2, 3};
6 }
```

Si l'on veut utiliser un tel code, on doit préciser explicitement le type de retour, comme un `std::vector` par exemple.

```
1 // Aucun problème.
2 auto create_initialisation_list() -> std::vector<int>
3 {
4     return {1, 2, 3};
5 }
```

2.2.3. Les lambdas génériques

Alors que les paramètres des lambdas se doivent d'être des types concrets en C++11 (c'est-à-dire `int`, `double`, etc), cette restriction est levée depuis C++14 et il est possible d'utiliser `auto` pour les paramètres. Comme pour les fonctions, `auto` utilise les règles de déduction des templates et donc le code suivant ne fonctionne tout simplement pas.

> For a generic lambda, the closure type has a public inline function call operator member template (14.5.2) whose template-parameter-list consists of one invented type template parameter for each occurrence of auto in the lambda's parameter-declaration-clause, in order of appearance.
> The invented type template-parameter is a parameter pack if the corresponding parameter-declaration declares a function parameter pack (8.3.5). The return type and function parameters of the function call operator template are derived from the lambda-expression's trailing-return-type and parameter-declaration clause by replacing each occurrence of auto in the decl-specifiers of the parameter-declaration-clause with the name of the corresponding invented template-parameter.
>
> *N4296 §5.1.2.5*

```
1 #include <utility>
2 #include <vector>
3
```

```cpp
int main()
{
    std::vector<int> v;

    auto reset = [&v](const auto & new_value) { v = new_value; };

    /*
        GCC Error: no match for call to '(main()::<lambda(const auto:1&)>) (<brace-enclosed initializer list>)'.
                Note:   template argument deduction/substitution failed: couldn't deduce template parameter 'auto:1'.

        Clang   Error: no matching function for call to object of type '(lambda at prog.cc:8:18)'.
                    Note: candidate template ignored: couldn't infer template argument ''.
    */
    reset({1, 2, 3});

    return 0;
}
```

Au contraire du suivant qui fait ce qu'on lui demande.

```cpp
#include <utility>
#include <vector>

int main()
{
    std::vector<int> v;

    auto good = [&v](const std::initializer_list<int> & list) { v = list; };

    good({1, 2, 3});

    return 0;
}
```

2.3. Résumé

Nous avons vu beaucoup de choses, mais les règles de `auto` sont simples à comprendre, puisque quasiment identiques aux templates. Il est temps d'écrire la règle clairement pour conclure cette partie.

Dans le cas d'une déclaration de variable, `auto` diffère des règles de templates sur les formes `auto x = {}` et `auto x{}` en C++11 et C++14 puisque ces formes entraînent la création de `std::initializer_list`.
Dans le cas de C++17, seul la forme `auto x = {}` entraîne la création d'un `std::initializer_list`.
Dans le cas d'une fonction ou d'une lambda, `auto` utilise **exactement les mêmes règles** que les templates.

3. Mais qu'est-ce que decltype ?

L'autre mot-clef utile pour l'inférence de type, le petit nouveau de 2011, c'est `decltype`. Il diffère de `auto` en ce que `decltype` conserve la présence d'éventuelles références ou de `const`.

3.1. Exemples d'utilisation

Un exemple bien connu car souvent repris pour illustrer l'intérêt de `decltype` est la création d'une fonction appliquant `operator+` à deux objets. Alors qu'une somme de deux `int` produit un résultat de type `int`, l'addition d'un `double` et d'un `int` donne au final un `double`. Écrire ce genre de fonction devient très simple quand on a le pouvoir de `decltype`.

```
1  template<typename Left, typename Right>
2  auto add(Left lhs, Right rhs) -> decltype(lhs + rhs)
3  {
4      return lhs + rhs;
5  }
6
7  int main()
8  {
9      // Sera de type int et vaudra 11.
10     auto integer_sum = add(5, 6);
11
12     // Sera de type double et vaudra 11.5.
13     auto floating_sum = add(5, 6.5);
14
15     return 0;
16 }
```

On peut aussi imaginer une fonction comparant deux objets afin de déterminer le plus petit et le renvoyer. Là encore, nous aimerions que le type de l'objet renvoyé soit exactement le même que le type de l'objet le plus petit. Avec `decltype`, aucun problème.

```cpp
template<typename Left, typename Right>
auto min(Left lhs, Right rhs) -> decltype(lhs < rhs ? lhs : rhs)
{
    return (lhs < rhs) ? lhs : rhs;
}

int main()
{
    // Sera de type int et vaudra 5.
    auto integer_min = min(5, 6);

    // Sera de type double et vaudra 3.5.
    auto floating_min = min(5, 3.5);

    return 0;
}
```

i

Dans nos deux exemples, la présence de `auto` est **purement syntaxique**. Elle signifie «T'inquiète, le type de retour est déduit après» et, en effet, le type de retour est bien celui déduit par `decltype`. Nous sommes obligés de le mettre à la fin et non au début, car `decltype` utilise les paramètres de la fonction et doit donc se trouver **après** leur déclaration.

Examinons un autre exemple tiré d'un article de Scott Meyers. Imaginons un code qui appelle `operator[]` sur un conteneur. Mais si cet opérateur appliqué sur un `std::vector<int>` retourne un `int&`, ce même opérateur appliqué à un `std::vector<bool>` retourne un `std::vector<bool>::reference`, qui n'est non pas un `bool&` mais un objet un peu spécial dû à la nature particulière de `std::vector<bool>` (plus d'informations sur StackOverflow).

Nous voulons donc un code qui s'adapte au type de retour, en prenant en compte la présence éventuelle de référence. Le code, bien qu'un peu complexe, est ci-dessous.

```cpp
#include <vector>

template <typename Container, typename Index>
auto grab(ContainerType && container, IndexType && index) ->

        decltype(std::forward<ContainerType>(container)[std::forward<IndexType>
{
    return
        std::forward<ContainerType>(container)[std::forward<IndexType>(index)]
}

int main()
{
```

```
12      std::vector<int> int_vector {1, 2, 3};
13      std::vector<bool> bool_vector {true, false, true};
14
15      // Sera de type int&.
16      decltype(grab(int_vector, 1)) int_grab = grab(int_vector, 1);
17
18      // Sera de type std::vector<bool>::reference.
19      // GCC : std::_Bit_reference.
20      // Clang : std::__1::__bit_reference<std::__1::vector<bool,
            std::__1::allocator<bool> >, true>
21      decltype(grab(bool_vector, 1)) bool_grab = grab(bool_vector,
            1);
22
23      return 0;
24  }
```

3.2. Le petit plus de C++14

Bon, `decltype`, ça marche très bien, mais c'est un peu redondant d'avoir à écrire deux fois la même chose, surtout comme dans le cas de `grab`, où le type de retour est long. Nous sommes des faignants, donc depuis C++14 nous disposons d'un moyen très simple d'écrire plus vite: `decltype(auto)`.

```
1  template <typename Container, typename Index>
2  auto grab(ContainerType && container, IndexType && index) ->
       decltype(auto)
3  {
4      return
           std::forward<ContainerType>(container)[std::forward<IndexType>(index)]
5  }
```

Forme qui peut être réduite encore un peu plus.

```
1  template <typename Container, typename Index>
2  decltype(auto) grab(ContainerType && container, IndexType && index)
3  {
4      return
           std::forward<ContainerType>(container)[std::forward<IndexType>(index)]
5  }
6
7  int main()
8  {
9      std::vector<int> int_vector {1, 2, 3};
10     std::vector<bool> bool_vector {true, false, true};
```

```
11
12      // Et cette nouvelle syntaxe peut s'utiliser ici aussi.
13      decltype(auto) int_grab = grab(int_vector, 1);
14      decltype(auto) bool_grab = grab(bool_vector, 1);
15
16      return 0;
17  }
```

La combinaison de `auto` et `decltype` signifie que l'on va déduire automatique le type de retour sans le préciser, d'où `auto`, en utilisant les règles de déduction de `decltype`.

3.3. La règle exacte de `decltype`

Nous avons vu des exemples, maintenant il serait bien d'apprendre le fonctionnement exact et précis de `decltype`. La règle qui suit est tirée de la norme C++ N4296 7.1.6.2 §4 .

Pour toute expression `decltype(e)`, le type déduit dépend de quatre cas de figure.

- Si `e` est une ***id-expression*** **non parenthésée** ou **un accès non parenthésé à un membre d'une classe**, alors `decltype(e)` est du même type que l'objet désigné par `e`.
- Si `e` est une **lvalue**, alors `decltype(e)` est de type `T&` où `T` est le type de `e`.
- Si `e` est une **xvalue**, alors `decltype(e)` est de type `T&&` où `T` est le type de `e`.
- Sinon, `e` est une **prvalue** et `decltype(e)` vaut simplement `T`.

> For an expression e, the type denoted by decltype(e) is defined as follows :
> - if e is an unparenthesized id-expression or an unparenthesized class member access (5.2.5), decltype(e) is the type of the entity named by e. If there is no such entity, or if e names a set of overloaded functions, the program is ill-formed ;
> - otherwise, if e is an xvalue, decltype(e) is T&&, where T is the type of e ;
> - otherwise, if e is an lvalue, decltype(e) is T&, where T is the type of e ;
> - otherwise, decltype(e) is the type of e.
>
> *N4296 7.1.6.2 §4*

Examinons en détails chaque point de la liste et nous verrons que la règle est simple à comprendre. On peut s'aider de la documentation ainsi que de StackOverflow .

3.3.1. *Id-expression* et membres de classe

C'est le plus simple: si `e` est le nom d'une variable, locale ou globale, ou bien le nom d'un membre d'une quelconque classe, alors `decltype(e)` renvoie le type de cette variable.

```cpp
struct A
{
    int i;
    const char * ptr;
};

int main()
{
    int a = 0;
    decltype(a); // Donne int.

    const int b = a;
    decltype(b); // Donne const int.

    int & c = a;
    decltype(c); // Donne int&.

    const int & d = a;
    decltype(d); // Donne const int&.

    A my_struct = {0, nullptr};
    decltype(my_struct.i);   // Donne int.
    decltype(my_struct.ptr); // Donne char const*.

    return 0;
}
```

3.3.2. Les lvalues

Une **lvalue** peut être vue comme un objet dont on peut prendre l'adresse, mais qui ne peut être déplacé. Mais pas que. Ainsi, une indirection (`*p`), une fonction retournant une *lvalue reference* (comme `std::getline`), un cast vers une *lvalue reference* (`static_cast<int&>(x)`), un accès à un tableau (`a[x]`) ou une expression parenthésée (`(x)`) sont aussi des lvalues. Dans ce cas, **le type déduit est `T&`**.

```cpp
struct A
{
    int i;
};

int main()
{
    int a = 0;
    decltype(a);    // Donne int.
    // Expression parenthésée.
```

```
11      decltype((a));  // Donne int& et non int.
12
13      int * ptr = &a;
14      decltype(ptr);  // Donne int*.
15      // Déréférencement de pointeur.
16      decltype(*ptr); // Donne int&, soit le type de a auquel on
            ajoute &.*/
17
18      int& foo();
19      // Fonction retournant une lvalue-reference.
20      decltype(foo()); // Donne int&.
21
22      // Chaîne littérale.
23      decltype("A C-string !"); // Donne char const (&)[13], une
            référence constante sur un tableau de 13 char.
24
25      int array[] = {0, 1, 2, 3};
26      // Accès à un tableau.
27      decltype(array[1]); // Donne int&.
28      return 0;
29  }
```

Si jamais tu avais l'habitude d'entourer de parenthèses le retour d'une fonction, il faudra se méfier maintenant.

```
1  template <typename T>
2  decltype(auto) function()
3  {
4      T obj;
5
6      // Retourne une référence sur un objet local !
7      return (obj);
8  }
```

3.3.3. Les xvalues

Une **xvalue** (*eXpiring values*) désigne un objet qui approche de sa fin de vie, qui peut être déplacé dans peu de temps (avec `std::move` par exemple). On les trouve notamment dans des expressions avec des *rvalue references*. Ainsi, une fonction retournant une *rvalue reference*, `std::move()` ou `static_cast<X&&>(x)` produisent des xvalues. Et pour ce genre d'expressions, **`decltype` déduit le type comme étant `T&&`**.

```cpp
int main()
{
    int&& foo();
    decltype(foo()); // Donne int&&.

    int x;
    decltype(std::move(x));            // Donne int&&.
    decltype(static_cast<int&&>(x));   // Donne int&&.

    return 0;
}
```

3.3.4. Les prvalues

Les **prvalues** (*pure rvalues*) sont assez simples à comprendre. En effet, ce sont des objets dont on ne peut prendre l'adresse mais qui peuvent être déplacés. Ainsi, les littéraux `3` et `2.71818` sont des prvalues de type `int` et `double`. De même, les objets créés par conversion implicite lors d'appel à des constructeurs sont des prvalues. Sont des prvalues également les fonctions qui ne renvoient pas de référence.

```cpp
void function_on_string(std::string && s);

// "Hello you !" est implicitement converti dans une std::string
// temporaire qui sera passée en argument à la fonction.
// La std::string temporaire est donc une prvalue.
function_on_string("Hello you !");
```

Dans ces cas là, **le type déduit est tout simplement `T`**.

```cpp
#include <string>

std::string function();

int main()
{
    // 42 est une prvalue de type int, ainsi donc sera integer.
    decltype(42) integer = 0;

    // std::string{} est une prvalue, donc str sera une simple
        std::string.
    decltype(std::string{}) str {"Hello"};

```

```
13      // function() est une prvalue, so other_str sera une simple
            std::string.
14      decltype(function()) other_str {"World"};
15
16      return 0;
17  }
```

3.4. Résumé

Ouf, c'est un poil plus complexe que pour `auto`, n'est-ce-pas? Comme le disait Scott Meyers dans une de ses présentations , il n'est pas nécessaire de connaître les règles de `decltype` de manière aussi précise.

> *i* Dans 99% des cas, il suffit de se souvenir que `decltype` conserve la présence de `const` et/ou de références.

4. Connaître et afficher le type exact

C'est vrai que connaître les règles est utile, mais il est des fois où l'on aimerait bien faire avouer au compilateur quel est le type exact qu'il a déduit. Et puis y'a-t-il parmi les lecteur des gens suspicieux qui aimeraient bien vérifier mes dires. Soit, vérifions. Nous règlerons nos comptes après. 🤘

4.1. Vérifier de façon statique

Si l'on désire connaître les types des objets passés en paramètres à une fonction, on peut utiliser la macro `__PRETTY_FUNCTION__`, utilisable avec GCC et Clang. Particulièrement utile avec les templates.

```
1   #include <iostream>
2   #include <string>
3
4   template <typename T>
5   void function(T && param)
6   {
7       std::cout << __PRETTY_FUNCTION__ << "\n";
8   }
9
10  int main()
11  {
12      function(42);
```

```
13      function(std::string{});
14
15      const int a = 0;
16      function(a);
17
18      return 0;
19  }
```

L'autre moyen efficace est de **provoquer volontairement une erreur de templates**, car le compilateur ne résistera pas à l'envie de vous cracher l'erreur à la figure avec, inclue, les types déduits.

```
1   #include <string>
2   #include <utility>
3
4   template <typename T>
5       class TD;
6
7   template <typename T>
8   void function(T & param)
9   {
10      TD<T> templateType;
11      TD<decltype(param)> paramType;
12  }
13
14  int main()
15  {
16      std::string s {"Hello world"};
17      // Mais de quel type peut bien être str ?
18      decltype(auto) str = std::move(s);
19
20      // Clang : implicit instantiation of undefined template
            'TD<std::__1::basic_string<char> >' TD<T> templateType;
21      // error: implicit instantiation of undefined template
            'TD<std::__1::basic_string<char> &>' TD<decltype(param)>
            paramType;
22      function(str);
23      // GCC : error: 'TD<std::__cxx11::basic_string<char> >
            templateType' has incomplete type : TD<T> templateType;
24      // error: 'TD<std::__cxx11::basic_string<char>&> paramType' has
            incomplete type TD<decltype(param)> paramType;
25
26      return 0;
27  }
```

4.2. Vérifier de façon dynamique

Pour vérifier pendant l'exécution, on voit certains codes utiliser `typeid`, mais les types retournés ne sont pas forcément clairs et leur lisibilité dépend du compilateur. Le meilleurs moyen qui existe pour avoir une solution portable et claire est **Boost**, avec l'en-tête `<boost/type_index.hpp>`. En plus, c'est un en-tête qui ne nécessite pas d'être compilé.

```cpp
#include <boost/type_index.hpp>
#include <iostream>
#include <memory>
#include <string>
#include <vector>

#define TYPE(T)
    boost::typeindex::type_id_with_cvr<T>().pretty_name()
#define TYPE_EXPR(expr)
    boost::typeindex::type_id_with_cvr<decltype(expr)>().pretty_name()

template <typename T>
void function(T && param)
{
    std::cout << "T = " << TYPE(T) << "\n";
    std::cout << "param = " << TYPE_EXPR(param) << "\n";
    std::cout << std::endl;
}

int main()
{
    /*
     * GCC
     * T = double
     * param = double&&
     *
     * Clang
     * T = double
     * param = double&&
     */
    function(3.1415926);

    /*
     * GCC
     * T = char const (&) [20]
     * param = char const (&) [20]
     *
     * Clang
     * T = char const (&) [20]
     * param = char const (&) [20]
     */
    function("Hello with C-string");
```

```
41
42      /*
43       * GCC
44       * T = std::unique_ptr<int, std::default_delete<int> >&
45       * param = std::unique_ptr<int, std::default_delete<int> >&
46       *
47       * Clang
48       * T = std::__1::unique_ptr<int, std::__1::default_delete<int>
        >&
49       * param = std::__1::unique_ptr<int,
        std::__1::default_delete<int> >&
50       */
51      auto ptr = std::make_unique<int>(42);
52      function(ptr);
53
54      /*
55       * GCC
56       * T = std::__cxx11::basic_string<char, std::char_traits<char>,
        std::allocator<char> > const&
57       * param = std::__cxx11::basic_string<char,
        std::char_traits<char>, std::allocator<char> > const&
58       *
59       * Clang
60       * T = std::__1::basic_string<char,
        std::__1::char_traits<char>, std::__1::allocator<char> > const&
61       * param = std::__1::basic_string<char,
        std::__1::char_traits<char>, std::__1::allocator<char> > const&
62       */
63      const std::string str("Hello with C++ std::string");
64      function(str);
65
66      /*
67       * GCC
68       * T = std::vector<float, std::allocator<float> > const&
69       * param = std::vector<float, std::allocator<float> > const&
70       *
71       * Clang
72       * T = std::__1::vector<float, std::__1::allocator<float> >
        const&
73       * param = std::__1::vector<float, std::__1::allocator<float> >
        const&
74       */
75      const std::vector<float> v;
76      function(v);
77
78      /*
79       * GCC
80       * T = main::{lambda()#1}&
81       * param = main::{lambda()#1}&
82       *
```

```
83          * Clang
84          * T = main::$_0&
85          * param = main::$_0&
86          */
87         auto lambda = []() -> int { return 42; };
88         function(lambda);
89
90         return 0;
91     }
```

5. Quand les utiliser ?

Tu ne sais pas penser de ces nouveautés et tu es tout perdu? Pour t'aider, voici des avis de différents programmeurs, récoltés sur Internet.

> Principalement pour les types moches et à rallonge, avec plein de templates dedans. De temps à autre je tente du AAA, mais sans trop me forcer à l'utiliser.
>
> *Luthaf*

> — `auto`: je l'utilise quand le type est très long, souvent avec la ST(L) et ses noms template à rallonge en retour de fonction. Mais quand je sais ce que je vais manipuler bien sûr (des itérateurs, conteneurs, etc.). Je l'utilise aussi pour des types numériques qui peuvent varier dans le temps (passer de float à double par exemple), ça permet de gagner pas mal de temps!
> — `decltype`: je l'utilise moins que le précédent mais lorsque je m'en sers c'est souvent avec auto (pas decltype(auto)), pour bien montrer que le type d'une variable doit absolument être le même que celui d'une autre.
>
> *zeFresk*

> Sinon, je mets `auto` quand la variable est initialisée avec une autre variable ou avec un retour de fonction. Dans les autres cas, j'appelle directement le constructeur `T x{...};` et non pas `auto x = T{...}`. Aussi dans les boucles sur intervalle (sauf si l'IDE décide de ne pas reconnaître le type… -_-).
> Je ne le mets pas quand je veux une interface. À la place, je mets le type de l'interface.
> Je ne l'utilise pas quand il y a `std::reference_wrapper`, sinon il faut mettre machin.get() partout. Je trouve ça regrettable en fait, j'espère que la proposition de surcharge de l'opérateur . va être accepté (pas du tout suivit le truc). (D'ailleurs, je remplace souvent reference_wrapper par à un proxy rien que pour cette raison…)
> `decltype` quand j'ai besoin de construire une variable du même type. Généralement, dans un alias (`using Truc = decltype(machin)`).
>
> *jo_link_noir*

> Salut! Mon avis:
> — `auto`: souvent pour les types qui peuvent changer (`float`, `double` notamment), presque toujours pour les variables initialisées par un retour de fonction (`make_shared`, `make_unique`, `begin` pour ne donner que des exemples de la SL);
> — `decltype`: dans des arguments de fonction qui ont deux fois le même type (exemple: `maFonction(UnTypeComplique::iterator first, decltype(first) last)`), rarement en d'autres circonstances
> — `decltype(auto)`: je n'ai jamais rencontré un cas de figure ou j'ai eu à l'utiliser, et je préfère l'éviter car je le trouve peu explicite, car il faut aller voir quels qualificateurs (`const`, `volatile`, référence) marquent la "variable source", ce qui est peu lisible. Je préfère dans ce cas réécrire `auto const&` par exemple.
>
> *mehdidou99*

> Les `auto`, c'est bien, mangez-en.
>
> *gbdivers*

> Personnellement, je suis dans la même optique que **@jo__link__noir**, j'utilise `auto` quand je crée un élément qui est dépendant d'un autre. Pour `decltype`, j'ajouterai une autre petite utilisation que pour la création d'un alias: une dépendance de type mais où la première utilisation ne nous donne pas l'info. Cas typique:
>
> ```
> 1 std::vector<bidule> v;
> 2
> 3 //...
> 4
> 5 for(decltype(v)::size_type i = 0; i < v.size(); ++i){
> 6
> 7 }
> ```
>
> Après, dans ce cas, on aura effectivement envie de définir le type avant avec un using.
>
> *Ksass'Peuk*

> Je les utilise quand ils permettent de gagner en concision sans perdre en clarté. Quand le type d'une expression est sans ambiguïté, je n'ai aucun soucis à utiliser `auto`. Si je veux m'adapter à la présence ou non d'un const, alors je combine `decltype` et `auto`.
>
> *Moi-même*

> — Use `auto` if a reference type **would never be correct**.
> — Use `decltype(auto)` only if a reference type could be correct.
>
> *Scott Meyers*

— Un peu de StackOverflow .
— Encore un peu plus .
— Et du cplusplus.com aussi.

Hé hé, qui pouvait penser qu'il y aurait autant à dire sur l'inférence de type en C++? Les règles restent néanmoins assez simples à comprendre, même si `decltype` demande, pour être bien compris, des connaissances plus poussées des nouveautés de C++ et notamment ces histoires de *values*.

En attendant, nous nous séparons ici. Si vous avez des questions, n'hésitez pas à les poser ici, sur Zeste de Savoir, **après avoir fait un minimum de recherche**, bien entendu.

Sur ce, à très bientôt!

Liste des abréviations

AAA Almost Always Auto. 27