COMP 5421/1 BB C++ auto Type Deduction July 3, 2018

1 Some examples

```
   std::map<std::string, std::vector<int>>::iterator it1;

   auto x = 5;              // type of x is int
   auto y = 5.0;            // type of y is double
   auto z = 5.0F;           // type of z is float
   auto a = 5L;             // type of a is long
   auto b = 1 + 1 == 11;    // type of b is bool
   auto c = 'c';            // type of c is char
   auto s = "auto";         // type of s is char*

   auto u { 5 };            // type of u is int (C++17)
   auto v = { 5 };          // type of v is std::initializer_list<int>

   //auto w{ 5 , 10 };      // error: more than one value in {},
                            //hence there must be an = before {
   auto w = { 5 , 10 };     // ok. type of w is sts::initializer_list<int>

   auto t = std::string("auto"); // s is of type std::string
   auto tt = t;                  // tt is type of std::string

   auto it2 = it1;
   // it2 is of type std::map<std::string, std::vector<int>>::iterator

   auto i = { 1, 2, 3 }; //  initializer_list<int>
   for (auto it = i.begin(); it != i.end(); ++it) //it is of type int*
   {
      cout << *it << endl;
   }
```

2 auto + multiple variables declaration

There can be as many declarations using a single auto provided that all declarations involve the same type:

```
int x = 10;
auto y = 10, &z = x, *p = &x;  // y is int, z is int&, p is int*

auto s = "auto", t{1.5};
// error: cannot initialize variables of different types using a single auto
```

3 auto Type Deduction Magic

Declaring new variables, the **auto** keyword directs the compiler to use an initializing expression to deduce the types of the variables it is declaring. Like any explicit type name, the **auto** keyword can be adorned with qualifiers such as const, const, *, &, and &&.

The rules followed by **auto** during a type deduction are not only dependent on **auto**'s adornments but also on the type of the initializing expression. These rules may be categorized as follows:

auto Adornments	var name	initializer	Type Deduction Steps
auto &	var =	expression	1. If the initializing expression is a reference, ignore the reference.
const auto & const auto auto	var =	expression	1. The same as above step 1. 2. If the initializing expression is now a top-level const, ignore the const too.
auto &&	var =	expression	1. The same as above step 1. 2. The same as above step 2. 3. If the initializing expression is an *lvalue*, then add an & reference qualifier to the type deduced so far.
auto *	var =	expression	1’. If the initializing expression is a pointer, ignore the pointer.
const auto *	var =	expression	1. The same as above step 1’. 2. The same as above step 2.

Note: Consider **auto** adorned with an *rvalue* reference **&&**, as in the middle category above, and suppose that the deduced type out of step 2 is **T**. Now, step 3 might produce **T&** as the deduced type, resulting in **auto &&** being replaced with **T& &&**. By the C++11 reference collapsing rules, **& &&** collapses to **&**, giving **T&** as the final deduced type for **var**.

For more information refer to auto, decltype, Universal References in C++11, and (View a free sample) of Overview of the New C++ (C++11/14).

4 More Examples

4.1 auto var = expression
const auto var = expression

```
int a = 11;                   // an int named a
auto a1 = a;                  // auto ≡ int, a1's type is int
const auto ca1 = a;           // auto ≡ int, ca1's type is const int

const int ca = a;             // a const int named ca
auto ca1 = ca;                // auto ≡ int, ca1's type is int
const auto ca2 = ca;          // auto ≡ int, ca2's type is const int

int b = 22;                   // an int named b

int  & rb = b;                // a reference to b
auto rb1 = rb;                // auto ≡ int, rb1's type is int
const auto rb1 = rb;          // auto ≡ int, rb1's type is const int

const int  & crb = b;         // a const reference to b
auto crb1 = crb;              // auto ≡ int, crb1's type is int
const auto crb2 = crb;        // auto ≡ int, crb2's type is const int
```

4.2 auto & var = expression

const auto & var = expression

```
int a = 11;               // an int named a
auto &  a1 = a;           // auto ≡ int, a1's type is int&
const auto &  a2 = a;     // auto ≡ int, a2's type is const int &

const int ca = a;         // a const int named ca
auto & ca1 = ca;          // auto ≡ const int, ca1's type is const int &
const auto & ca2 = ca;    // auto ≡ int, ca1's type is const int &

int b = 22;               // an int named b

int & rb = b;             // a refrence to b
auto & rb1 = rb;          // auto ≡ int, rb1's type is int &
const auto & rb2 = rb;    // auto ≡ int, rb2's type is const int &

const int & crb = b;      // a const refrence to b
auto &crb1 = crb;         // auto ≡ const int, crb1's type is const int &
const auto & crb2 = crb;  // auto ≡ int, crb2's type is const int &
```

4.3 auto * var = expression

const auto * var = expression

```
int a = 11;               // an int named a

int *pa = &a;             // a pointer named pa
auto * pa1 = pa;          // auto ≡ int, pa1's type is int *
const auto * pa2 = pa;    // auto ≡ int, a1's type is const int *

const int * cpa = &a;     // a const pointer named cpa
auto * cpa1 = cpa;        // auto ≡ const int, a1's type is const int *
const auto * cpa2 = cpa;  // auto ≡ int, a1's type is const int *
```

5 Using auto with an array initializer

```
   int a[10];
   auto b = a;        // b is of type int*

   auto& c = a;       // c is of type int(&)[10]
   int(&d)[10] = a;   // d is of type int(&)[10]

   int aa[11];
   auto& cc = aa;     // cc is of type int(&)[11]
   int(&dd)[10] = aa; // this does not compile (why?)
```

6 Using auto with a function initializer

```
double fun(int x) { return 1.0 * x; }
```

```
   auto f = fun;         // f is of type double(*)(int)
   cout << f(5) << endl;

   auto& g = fun;        // g is of type double(&)(int)
   cout << g(5) << endl;

   auto* h = fun;        // h is of type double(*)(int)
   cout << h(5) << endl;
   // the compiler automatically dereference h to (*h)
   cout << (*h)(5) << endl;
   // simply use h(5), the compiler performs the dereference
```