Morden cpp note — auto

Thursday, December 3, 2020 4:04 PM

- auto type deduction.

因为C++是一种静态强类型的语言，任何变量都要有一个确定的类型，否则就不能用。这在变量类型简单的时候还好说，eg. int, double，但在泛型编程的时候，麻烦就来了。泛型编程有很多模板参数，有的类型还有内部子类型，把C++的类型体系搞复杂了。

```
int  i = 0;  // 整型变量，类型容易知道.
double  x = 1.0;  // 浮点数变量，类型容易知道.

std::string str = "hello"; // 字符串变量，有了名字空间，麻烦了一些.

std::map<int, std::string> m = {{1, "a"}, {2, "b"}}; // 使用初始化列表的形式.
// 关联数组，名字空间加模版参数，很麻烦.

std::map<int, std::string>::const_iterator  iter = m.begin();
// 内部子类型，超级麻烦.

??? = bind1st(std::less<int>(), 2); // 根本写不出来.
```

关键字auto，在代码里的作用像是 placeholder，编译器会自动填上正确的类型.

```
auto  i = 0; // 自动推导为 int 类型.
auto  x = 1.0; // 自动推导为double 类型.

auto  str = "hello"; // 自动推导为 const char [6] 类型.

std::map<int, std::string> m={{1,"a"},{2,"b"}}; // 自动推导不出来.

auto  iter = m.begin(); // 自动推导为map内部的迭代器类型.

auto  f = bind1st(std::less<int>(), 2); // 自动推导出类型，具体类型不知道.
```

需要注意，C++太复杂，自动类型推导有时候可能失效，比如把字符串类型推导为了const char [6]，而不是 std::string.
除了简化代码，auto还避免了对类型的硬编码，eg. 把map改为 unordered_map，后面代码全不用动.

auto只能用在初始化时，包括赋值初始化和花括号初始化（初始化列表，initializer list），变量右边必需要有一个表达式，如果不是初始化的形式，只是纯变量声明，就无法使用auto，因为没有表达式让auto推导.

```
auto  x = 0L;   // 自动推导为 long
auto  y = &x;   // 自动推导为 long*
auto  z {&x};   // 自动推导为 long*.
auto  err; // 错误，没有赋值表达式，不知道是什么类型.
```

还有个特殊情况，在类成员变量初始化时，C++不允许使用auto，

```
class  X  final
{
    auto  a  =  10; // 错误，类里不能使用 auto
};
```

auto推导规则：

- auto总是推导出"值类型"，绝不会是"引用".
- auto可以附加上const，volatile，*，&这样的类型修饰符，得到新类型.

```
auto  x = 10L;  // auto推导为 long，x是 long.
auto &  x1 = x;  // auto 推导为 long，x1 是 long&.
auto*   x2 = &x; // auto推导为 long，x2是 long*.
const  auto &  x3 = x; // auto推导为 long，x3 是 const long&.
auto  x4 = &x3; // auto推导为 const long*，x4是 const long*.
```

C++的自动类型推导，还有 decltype，形式像函数，后面圆括号里就是可用于计算类型的表达式（和 sizeof 类似），其它方面和 auto 一样，因为它自带表达式，不需要变量后面再有表达式，可以直接声明变量.

```
int  x = 0;   // 整型变量
decltype(x)  x1;  // x1是 int
decltype(x) &  x2 = x; // 推导为int，x2是int&，引用必须赋值.
decltype(x)*  x3; // 推导为int，x3是 int*.
decltype(&x)  x4; // x4是 int*
decltype(&x)*  x5; // 推导为int*，x5是int**.
decltype(x2)  x6 = x2; // 推导为int&，x6是int&，引用必须赋值.
```

decltype 不仅能推导出值类型，还能推导出引用类型，也就是表达式的原始类型.
在示例代码中，除了加上 *，& 修饰，decltype还可以直接从一个引用类型的变量推导出引用类型，而auto就会把引用去掉，推导出值类型.
所以，完全可以把decltype看成是一个真正的类型名，用在变量声明，函数参数/返回值，模板参数等任何类型出现的地方，只不过这个类型是通过表达式推导得到。

```
Eg. using  int_ptr = decltype(&x); // int*
```

decltype缺点是写起来略麻烦，特别是在初始化的时候，表达式要重复两次，左边的类型计算，右边的初始化，把简化代码的优势抵消了。所以，C++14增加了decltype(auto)形式，即可以精确推导类型，又像auto一样方便.

```
int  x = 0; // 整型变量
decltype(auto)  x1 = (x); // 推导为 int&，因为(expr)是引用类型.
decltype(auto)  x2 = &x; // 推导为 int*
decltype(auto)  x3 = x1; // 推导为int&.
```

因为auto写法简单，在变量声明时应该尽量多用 auto.

auto 还有一个最佳应用，就是"range-based for"，为了保证效率，最好使用 const auto& 或 auto&.

```
vector<int>  v = {2, 3, 5, 7, 11};

for (const auto&  i : v) {  // 常引用方式访问元素，避免拷贝代价.
    cout << i << ",";    // 常引用不会改变元素.
}

for (auto&  i : v) {
    i++;                  // 引用方式访问元素.
    cout << i << ",";     // 可以改变元素的值.
}
```

在C++14中，auto新增了一个用途，就是可推导函数的返回值，这样在写复杂函数的时候，比如返回一个pair，容器或迭代器，就会省事.

```
auto  get_a_set()   // auto作为函数返回值的占位符.
{
   std::set<int>  s = {1, 2, 3};
   return  s;
}
```

decltype：当需要特殊类型时使用，比如，定义函数指针在C++里比较头疼，用 decltype 就可以很容易得到指针类型。

```
// UNIX信号函数原型
void  (*signal(int signo, void (*func)(int)))(int)
// 使用decltype可以轻松获得函数指针类型
using  sig_func_ptr_t = decltype(&signal);
```

在定义类的时候，因为auto被禁用，可以用decltype搭配别名任意定义类型，再应用到成员变量，成员函数上，变通地实现auto

```
class  DemoClass  final
{
public:
    using  set_type  =  set::set<int>; // 集合类别名.

private:
    set_type  m_set; // 使用别名定义成员变量.
    // 使用decltype计算表达式的类型，定义别名.
    using  iter_type  = decltype(m_set.begin());
    iter_type  m_pos;  // 类型别名定义成员变量.
};
```

C++14新增了字面量后缀"s"，表示标准字符串，就可以用 auto str = "xxx"s; 的形式推导出std::string类型。