Modern cpp 30 lectures — auto

Sunday, December 20, 2020 8:12 AM

auto
自动类型推断,就是编译器能根据表达式的类型,自动决定变量类型,
还有函数的返回类型,

```
// vector<int> v;
for (vector<int>::iterator
        it = v.begin(),
        end = v.end();
        it != end; ++it) {
    //循环体
}
```

可改写为

```
for (auto it = v.begin(), end = v.end();
     it != end; ++it) {
    //循环体
}
```

不使用自动类型推断时,如果容器类型未知的话,还需要加上 typename

```
template <typename T>
void foo(const T& container)
{
  for (typename T::const_iterator
          it = v.begin(),
       ...
}
```

因为用了 const 引用,要求写 const_iterator 作为迭代器类型.

如果begin返回类型不是该类型的 const_iterator 嵌套类型的话,那实际上不用自动类型推断就没法表达了.
Eg. 遍历函数要求支持 C 数组的话,不用自动类型推断,就只能使用两个不同的重载:

```
template <typename T, std::size_t N>
void foo(const T (&a)[N])
{
  typedef const T* ptr_t;
  for (ptr_t it = a, end = a + N;
        it != end; ++it) {
   //循环体
  }
}

template <typename T>
void foo(const T& c)
{
  for (typename T::const_iterator
          it = c.begin(),
          end = c.end();
          it != end; ++it) {
     //循环体
  }
}
```

如果使用自动类型推导,再加上 C++11 提供的全局 begin 和 end 函数,
上面的代码可统一成:

```
template <typename T>
void foo(const T& c)
{
  using std::begin;
  using std::end;

  for (auto it = begin(c),
        ite = end(c);
        it != ite; ++it) {
    //循环体
  }
}
```

使用依赖参数查找 (ADL). 就是 begin 和 end,对象 c 所属类型所在的名空间里这两个函数将被优先使用.

auto 使用规则类似于函数模板参数的推导规则, Eg.
- auto a = expr; 意味着用 expr 去匹配一个假想的 template <typename T> f(T) 函数模板,结果为值类型
- const auto& a = expr; 意味着用 expr 去匹配一个假想的 template <typename T> f(const T&) 函数模板,结果为常左值引用类型.
- auto&& a = expr; 意味着用 expr 去匹配一个假想的 template <typename T> f(T&&) 函数模板,结果是一个跟 expr 值类别相同的引用类型.

decltype
用途是获得一个表达式的类型,
- decltype(变量名) 可以获得变量的精确类型. Eg. decltype(a) 会获得 int, 因为 a 是 int
- decltype(表达式) 可以获得表达式的引用类型,除非表达式的结果是个纯右值(prvalue),此时结果仍为值类型.
  - decltype((a)) 会获得 int&, 因为 a 是 lvalue.
  - decltype(a+a) 会获得 int, 因为 a+a 是 prvalue.

decltype(auto)
- auto 是值类型
- auto& 是左值引用类型
- auto&& 是转发引用(可以是左值引用,也可以是右值引用)
- decltype(expr) 既可以是值类型,也可以是引用类型.
  - 可写为 decltype(expr) a = expr; 但表达式很长的情况不适合.
  - C++14 引入了 decltype(auto), 对于上述情况,只需要写 decltype(auto) a = expr;

函数返回值类型推断.
从 C++14 开始,函数的返回值也可以用 auto 或 decltype(auto) 来声明了
- 用 auto 可以得到值类型
- 用 auto& 或 auto&& 可得引用类型
- decltype(auto) 可以根据返回表达式通用地决定返回的是值类型还是引用类型.

后置返回值类型声明

```
auto foo(参数) -> 返回值类型声明.
{
  //函数体
}
```

在返回类型比较复杂,特别是返回类型跟参数类型有某种推导关系时会使用.

类模板的模板参数推导.
Pair
- Pair 一般不会用 pair<int, int> pr{1, 42};
- pair 通常会用 auto pr = make_pair(1, 42);
- 在 C++17, 可以直接写 pair pr{1, 42};

array

```
int a1[] = {1, 2, 3};
array<int, 3> a2{1, 2, 3}; ← 有点儿繁琐
//array<int> a3{1, 2, 3}; ← 错误
```

在 C++17 中可直接写 array a{1, 2, 3}; ← 得到 array<int, 3>.

这种自动推导机制,可以是编译器根据构造函数来自动生成:

```
template <typename T>
struct MyObj {
  MyObj(T value);
  ...
};

MyObj obj1{string("hello")}; ← 得到 MyObj<string>
MyObj obj2{"hello"}; ← 得到 MyObj<const char*>
```

也可以手工推导

```
template <typename T>
struct MyObj {
  MyObj(T value);
  ...
};

MyObj(const char*) -> MyObj<string>;

MyObj obj{"hello"}; → 得到 MyObj<string>
```

完整代码:

```
#include <string>
using namespace std;
template <typename T>
struct MyObj {
  MyObj(T value)
    : value_(value) {}
  T value_;
}
MyObj(const char*) -> MyObj<string>;
int main()
{
  MyObj {"hello"};
}
```

const char* 就是 char const *, 都是指向常字符的指针. 指针指向的内容不可更改.
如果写成 char* const 是指向字符的常指针,指针本身不可更改,指向的内容可更改.

结构化绑定
在关联容器中的一个例子:

```
multimap<string, int>::iterator lower, upper;
std::tie(lower, upper) = mmp.equal_range("four");
```

返回值是个 pair, 我们希望用两个变量来接收数值,就不得不声明了两个变量,然后使用 tie 来接收结果.
C++17 中,可以写为

```
auto [lower, upper] = mmp.equal_range("four");
```

这个语法使得我们可以用 auto 声明变量分别获取 pair 或 tuple 返回值里各个子项,让代码可读性更好.

列表初始化
在 C++98 里,标准容器比起 C 风格数组有一个劣势,不能方便的初始化容器内容:
- 对于数组: int a[] = {1, 2, 3};
- 对于 vector 却得写:

```
vector<int> v;
v.push(1);
v.push(2);
v.push(3);
```

- 于是 C++ 引入了列表初始化,

```
vector<int> v{1, 2, 3};
```

统一初始化
使用大括号{}来进行对象的初始化,是 C++11 引入的新语法,能够代替很多小括号()在变量初始化时的使用,被称为统一初始化 uniform initialization.
大括号对于构造一个对象而言,最大的好处是避免了 C++ 里的语法分析 the most vexing parse, Eg.

```
class utf8_to_wstring {
public:
    utf8_to_wstring(const char*);
    operator wchar_t*();
}
```

如果在 Windows 下用这个类转换文件名,打开文件:

```
ifstream ifs(
    utf8_to_wstring(filename));
```

实际执行的是

```
ifstream ifs(
    utf8_to_wstring filename);
```

编译器认为是声明了一个叫 ifs 的函数,而不是对象.
如果把任意一对小括号替换成大括号,或都替换,可避免此类问题.

```
ifstream ifs{
  utf8_to_wstring{filename}};
```

几乎可以在所有初始化对象的地方使用大括号而不是小括号,还有一个附带的特点:当一个构造函数没有标成 explicit 时,可以使用大括号不写类名来进行构造, Eg.

```
Obj getObj()
{
  return {1.0};
}
```

如果 Obj 类可以使用浮点数进行构造的话,上面的写法就是合法的.
如果有无参数,多参数的构造函数,也可以使用这个形式.
除了形式上的区别,它跟 Obj(1.0) 的主要区别是,后者可以用来调用 Obj(int), 而使用大括号时编译器会拒绝"窄"转换,不接受以 {1.0} 或 Obj{1.0} 的形式调用构造函数 Obj(int).

这个语法的主要限制是,如果一个构造函数既有使用初始化列表的构造函数,又有不使用初始化列表的构造函数,那编译器会千方百计地试图调用使用初始化列表的构造函数,导致各种意外. 所以
- 如果一个类没有使用初始化列表的构造函数时,初始化该类对象可全部使用统一初始化语法.
- 如果一个类有使用初始化列表的构造函数时,则只应用在初始化列表构造的情况.

Eg. 如果一个类 obj 既有 Obj(initializer_list<int>); 又有 Obj(double); 想调用后面那个构造函数,就别用 Obj{1.0}, 而用 Obj(1.0)

类数据成员的默认初始化.
C++11 允许在声明数据成员时直接给予一个初始化表达式. 当且仅当构造函数的初始化列表中不包含该数据成员时,这个数据成员就会自动使用初始化表达式进行初始化.
E.g.

```
class Complex {
public:
  Complex()
    : re_(0), im_(0) {}

  Complex(float re)
    : re_(re), im_(0) {}

  Complex(float re, float im)
    : re_(re), im(_im) {}

  ...
private:
    float re_;
    float im_;
};
```

假设由于某种原因,不能使用缺省参数来简化构造函数,而使用数据成员的默认初始化,可写成:

```
class Complex()
public:
  Complex() {}

  Complex(float re): re_(re) {}

  Complex(float re, float im)
    : re_(re), im_(im) {}

private:
   float re_{0};
   float im_{0};
};
```

- Complex() {} ← 构造函数没有任何初始化列表,所以类数据成员的初始化全部由默认初始化完成. re_, im_ 都是 0.
- Complex(float re): re_(re) {} ← 构造函数提供了 re_ 的初始化, im_ 仍由默认初始化完成.
- Complex(float re, float im) : re_(re), im_(im) {} ← 构造函数完全不使用默认初始化.

Tips
- 把一个变量的完整类型信息打印出来.

```
#define TYPE_DISPLAY(var) \
     static type_displayer<decltype(var)> type_display_test

template <typename T> //declaration only for type_displayer
class type_displayer;
```

用的时候就写 TYPE_DISPLAY(变量名字).