3.1.10 Lambda

C++11 引入了 *lambda*，允许 inline 函数的定义式被用作一个参数，或是一个 local 对象。

Lambda 改变了 C++ 标准库的用法。比如，6.9 节第 229 页及 10.3 节第 499 页讨论了如何使用 lambda 搭配 STL 算法和容器，18.1.2 节第 958 页则展示了如何使用 lambda 定义并发（concurrently）代码。

Lambda 的语法

所谓 lambda 是一份功能定义式，可被定义于语句（statement）或表达式（expression）内部。因此你可以拿 lambda 当作 inline 函数使用。

最小型的 lambda 函数没有参数，什么也不做，如下：

```
[] {
    std::cout << "hello lambda" << std::endl;
}
```

可以直接调用它：

```
[] {
    std::cout << "hello lambda" << std::endl;
} ();          // prints ''hello lambda''
```

或是把它传递给对象，使之能被调用：

```
auto l = [] {
             std::cout << "hello lambda" << std::endl;
         };
...
l();           // prints ''hello lambda''
```

如你所见，lambda 总是由一个所谓的 *lambda introducer* 引入：那是一组方括号，你可以在其内指明一个所谓的 *capture*，用来在 lambda 内部访问“nonstatic 外部对象”。如果无须访问外部数据，这组方括号可以为空，就像本例所示。Static 对象，诸如 `std::cout`，是可被使用的。

在 lambda introducer 和 lambda body 之间，你可以指明参数或 `mutable`，或一份异常明细（exception specification）或 attribute specifier 以及返回类型。所有这一切都可有可无，但如果其中一个出现了，参数所需的小括号就必须出现。因此 lambda 语法也可以是

```
[...]  {...}
```

或

```
[...]  (...)  mutable_opt  throwSpec_opt  ->retType_opt  {...}
```

Lambda 可以拥有参数，指明于小括号内，就像任何函数一样：

```
auto l = [] (const std::string& s) {
             std::cout << s << std::endl;
         };
l("hello lambda");     // prints ''hello lambda''
```

然而请注意，lambda 不可以是 template。你始终必须指明所有类型。

Lambda 也可以返回某物。但你不需要指明返回类型，该类型会根据返回值被推导出来。例如下面的 lambda 的返回类型是 int：

```
[] {
    return 42;
}
```

如果一定想指明一个返回类型，可使用新式 C++ 语法，一般函数也可以用它（见 3.1.12 节第 32 页）。例如下面的 lambda 返回 42.0：

```
[] () -> double {
    return 42;
}
```

这么一来你就必须指明返回类型，放在实参所需的小括号以及字符 -> 之后。

在参数和返回类型指示（return specification）或函数体之间，你也可以写出一份异常明细（exception specification），就像你能够为一般函数所做的那样。然而目前已不鼓励为函数撰写异常明细（见 3.1.7 节第 24 页）。

Capture（用以访问外部作用域）

在 lambda introducer （每个 lambda 最开始的方括号）内，你可以指明一个 *capture* 用来处理外部作用域内未被传递为实参的数据：

- [=] 意味着外部作用域以 by value 方式传递给 lambda。因此当这个 lambda 被定义时，你可以读取所有可读数据（readable data），但不能改动它们。
- [&] 意味着外部作用域以 by reference 方式传递给 lambda。因此当这个 lambda 被定义时，你对所有数据的涂写动作都合法，前提是你拥有涂写它们的权力。

也可以个别指明 lambda 之内你所访问的每一个对象是 by value 或 by reference。因此你可以对访问设限，也可以混合不同的访问权力。例如下面这些语句：

```
int x=0;
int y=42;
auto qqq = [x, &y] {
                std::cout << "x: " << x << std::endl;
                std::cout << "y: " << y << std::endl;
                ++y;  // OK
            };
x = y = 77;
qqq();
qqq();
std::cout << "final y: " << y << std::endl;
```

会产生以下输出：

```
x: 0
y: 77
x: 0
y: 78
final y: 79
```

由于 x 是因 by value 而获得一份拷贝，在此 lambda 内部你可以改动它，但若调用 ++x 是通不过编译的。 y 以 by reference 方式传递，所以你可以涂写它，并且其值的变化会影响外部；所以调用这个 lambda 二次，会使指定值 77 被累加。

你也可以写 [=, &y] 取代 [x, &y]，意思是以 by reference 方式传递 y，其他所有实参则以 by value 方式传递。

为了获得 passing by value 和 passing by reference 混合体，你可以声明 lambda 为 mutable。下例中的对象都以 by value 方式传递，但在这个 lambda 所定义的函数对象内，你有权力涂写传入的值。例如：

```
int id = 0;
auto f = [id] () mutable {
             std::cout << "id: " << id << std::endl;
             ++id;    // OK
         };
id = 42;
f();
f();
f();
std::cout << id << std::endl;
```

会产生以下输出：

```
id: 0
id: 1
id: 2
42
```

你可以把上述 lambda 的行为视同下面这个 function object（见 6.10 节第 233 页）：

```
class {
  private:
    int id;  // copy of outside id
  public:
    void operator() () {
            std::cout << "id: " << id << std::endl;
            ++id;    // OK
    }
};
```

由于 mutable 的缘故，operator () 被定义为一个 non-const 成员函数，那意味着对 id 的涂写是可能的。所以，有了 mutable， lambda 变得 stateful，即使 state 是以 by value 方式传递。如果没有指明 mutable（一般往往如此），operator () 就成为一个 const 成员函数，那么对于对象你就只能读取，因为它们都以 by value 方式传递。 10.3.2 节第 501 页有一个例子使用 lambda 并使用 mutable，该处会讨论可能出现的问题。

Lambda 的类型

Lambda 的类型，是个不具名 function object（或称 functor）。每个 lambda 表达式的类型是独一无二的。因此如果想根据该类型声明对象，可借助于 template 或 auto。如果你实在需要写下该类型，可使用 decltype()（见 3.1.11 节第 32 页），例如把一个 lambda 作为 hash function 或 ordering 准则或 sorting 准则传给 associative （关联式）容器或 unordered （不定序）容器，详见 6.9 节第 232 页及 7.9.7 节第 379 页。

或者你也可以使用 C++ 标准库提供的 std::function<> class template，指明一个一般化类型给 functional programming（见 5.4.4 节第 133 页）使用。这个 class template 提供了“明确指出函数的返回类型为 lambda”的唯一办法：

```
// lang/lambda1.cpp

#include<functional>
#include<iostream>

std::function<int(int,int)> returnLambda ()
{
    return [] (int x, int y) {
                return x*y;
            };
}

int main()
{
    auto lf = returnLambda();
    std::cout << lf(6,7) << std::endl;
}
```

程序的输出当然是：

```
42
```