Morden cpp note — lambda

Monday, December 7, 2020 8:03 AM

C++不允许定义嵌套函数.

```
void my_square(int x)  //定义一个函数
{
  cout << x*x << endl;  //函数具体内容
}
auto pfunc = &my_square;  //只能用指针操作函数.
(*pfunc)(3);  //可以用*访问函数
pfunc(3);  //也可以直接调用函数指针.
```

所以，在面向过程编程范式里，函数和变量虽然是程序里最关键的两个部分，但却因为没有值，没有作用域而不能一致地处理函数.

lambda:

```
auto func = [](int x)  //定义一个lambda表达式.
{
  cout << x*x << endl;  // lambda表达式的具体内容.
};
func(3);  //调用lambda表达式.
```

重要的是，这个函数采用了赋值的方法，存入了一个变量，这是lambda函数与普通函数最大，最根本的区别.
因为lambda表达式是一个变量，所以可以随时随地在调用点定义函数，限制它的作用域和生命周期，实现函数的局部化.
而且因为lambda表达式和变量类似，能对它做各种运算，生成新的函数，就像数学里的复合函数.
lambda引出的全新思维方式是函数式编程，把计算看成是数学意义上的求解函数.

C++里的lambda表达式，除了可以像普通函数一样被调用，还有一个普通函数不具备的特殊本领，就是可以"捕获"外部变量，在内部的代码里直接操作.

```
int n = 10;  //一个外部变量
auto func = [=](int x)  //lambda表达式，用"="值捕获.
{
  cout << x*n << endl;  //直接操作外部变量.
};
func(3);  //调用lambda表达式.
```

函数式编程与命令式编程(面向过程)有很大不同，它像做数学那样运算计算，推导结果.

```
auto a = [](int x)  //a函数执行一个功能
  {...}
auto b = [](double x)  //b函数执行一个功能
  {...}
auto c = [](string str)  //c函数执行一个功能.
  {...}
auto f = [](...)  //f函数执行一个功能.
  {...}
return f(a,b,c)  //f调用a/b/c运算得到的结果
```

使用lambda的注意事项.

1. lambda的形式

C++没有为lambda引入新的关键字，而是用了一个特殊形式"[]"，术语叫"lambda引出符"，lambda introducer.
在lambda introducer后面，就可以像普通函数那样，用圆括号声明入口参数，用花括号定义函数体.
E.g.

```
auto f1 = [](){};  //空函数.
```

实际开发中，lambda表达式一定要有良好的缩进格式，必要时可以用注释来强调.

```
auto f2 = []()  //定义一个lambda表达式.
{
  cout << "lambda f2" << endl;
  auto f3 = [](int x)  //嵌套定义lambda表达式
  {
    return x*x;
  };// lambda f3  //使用注释显式说明表达式结束

  cout << f3(10) << endl;
}; // lambda f2  //使用注释显式说明表达式结束.
```

在lambda表达式赋值的时候，总是用auto来推导类型，因为每个lambda都有一个独特类型，无法直接写出来，必须用auto.

因为lambda表达式毕竟不是普通变量，所以尽量匿名使用lambda表达式，不必显示赋值给一个有名字的变量，直接声明就能用，免去费力起名的烦恼，因为匿名，lambda调用完后就不存在了，最小化它的影响范围，让代码更安全.

```
vector<int> v = {1,2,3,4,5};  //标准容器.
cout << *find_if(begin(v), end(v),  //标准库里的查找算法.
          [](int x)  //匿名lambda表达式，不需要auto赋值.
          {
            return x >= 5;  //用作算法的谓词判断条件
          }  //lambda表达式结束.
        )
     << endl;  //语句执行完，lambda表达式就不存在了.
```

2. lambda的变量捕获.

- [=]表示按值捕获所有外部变量，表达式内部是值的拷贝，并且不能修改.
- [&]是按引用捕获所有外部变量，内部以引用的方式使用，可以修改.

```
int x = 33;  //一个外部变量
auto f1 = [=]()  //lambda表达式，用"="按值捕获.
{
  // x += 10;  //x只读，不允许修改.
};
auto f2 = [&]()  //lambda表达式，用"&"按引用捕获.
{
  x += 10;  //x是引用，可以修改.
};
auto f3 = [=, &x]()  //lambda表达式，用"&"按引用捕获x，其他的按值捕获.
{
  x += 20;  //x是引用，可以修改.
};
```

外部变量的含义，是在lambda表达式定义之前，所有出现的变量，不管是局部的还是全局的. 对于就地使用的小lambda表达式，可以用[&]，而对于非本地调用，生命周期较长的lambda表达式，应慎用[&]捕获引用，而且最好在[]里显式地写出变量列表.

```
class DemoLambda final
{
  private:
    int x = 0;

public:
  auto print()  //返回一个lambda表达式供外部使用.
  {
    return [this]()  //显式捕获this指针
    {
      cout << "member = " << x << endl;
    };
  }
};
```

3. 泛型的lambda.

在C++14里，lambda表达式又多了一项，可以实现"泛型化"，相当于简化了的模板函数.

```
auto f = [](const auto& x)  //参数使用auto声明，泛型化.
{
  return x + x;
};
cout << f(3) << endl;  //参数类型是int.
cout << f(0.618) << endl;  //参数类型是double.
string str = "matrix";
cout << f(str) << endl;  //参数类型是string.
```

这个新特性在写泛型函数的时候非常方便，摆脱了冗长的模板参数和函数参数列表，可以尝试在今后的代码里多用lambda代替普通函数，能少写代码.

Tips:

- lambda可以视作智能函数，价值体现在就地定义，变量捕获等能力上.
- 用map + lambda 来替代难以维护的if/else/switch，可读性比大量的分支语句好得多.
- [=]是按值捕获，[&]是按引用捕获，[]则是无捕获，相当于普通函数.

更多的例子:

- map+lambda.

```
map<int, function<void()>> funcs;
funcs[1] = [](){...};
funcs[7] = [](){...};
funcs[42] = [](){...};
return funcs[x]();
```

这样，就把switch/case语句转换成了function+lambda，在map里的switch.

lambda.cpp 的demo.
运行:

```
g++ lambda.cpp -std=c++14 -o a.out
g++ lambda.cpp -std=c++14 -I../common -o a.out
```

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

//demo for function + lambda
class Demo final
{
public:
  using func_type = std::function<void()>;

public:
  func_type print = [this]()
  {
    cout << "value = " << m_value << endl;
    cout << "hello function + lambda " << endl;
  };

private:
  int m_value = 10;
};
int main()
{
  Demo d;
  d.print();
}
```

While a lambda doesn't have to be used with STL, this is by far the most common purpose.
E.g. int_list is a std::list of integers, and you want to print them out in an unusual "custom" function,

```
// define a special purpose custom printing function.
void print_it(int i)
{
  cout << ":" << i << ":";
}
...
// apply print_it to each integer in the list.
for_each(int_list.begin(), int_list.end(), print_it);
```

However, if print_it is never used anywhere else, it seems a bad idea to use this special case function at the same level with other general functions. So we need to use lambda to change to

```
for_each(int_list.begin(), int_list.end(), [](int i){cout << ":" << i << ":";});
```

`[](int i){cout << ":" << i << ":";}` ——
It starts with the square open/close brackets, followed by a function-like parameter list and a function-like body in curly braces.
The lambda is in the function slot of for_each call, meaning that you can place a lambda anywhere you place a function name.
In effect, you get to write the function "in place" in the code that calls it, instead of declaring it and defining it seperately.
The lambda behaves like a function in the for_each, but it did not have to be declared or defined anywhere else, and it doesn't have a name, and does not exist outside the scope of the for_each.

Why it is called lambda? In mathematical recursive function theory, λ is used to denote a function.

A lambda expression creates a thing that can be saved and treated like a function pointer or function object, this is how lambda is used in for_each.
We could store the lambda in a variable, and then call it using the syntax that we would also use a function pointer or function object:

```
auto func1 = [](int i){cout << ":" << i << ":";};
func1(42);
```

We can also return value from lambda:

```
if([](int i, int j){return 2*i == j;}(12, 24))
  cout << "j is 2 times of i" << endl;
else
  cout << "j is NOT 2 times of i" << endl;
```

We can also specify the return type:

```
cout << "This lambda returns " <<
  [](int x, int y) -> int {
    if (x>5)
      return x+y;
    else if (y<2)
      return x-y;
    else
      return x*y;
  }
  (4,3) << endl;
```

lambda can capture the variables outside: by value

```
{
  int int_var = 42;
  double dbl_var = 3.14;
  [int_var, dbl_var] ()
  {
    int i = 7;
    cout << int_var << ' ' << dbl_var << ' ' << i << endl;
  } ();
}
```

or by reference:

```
int int_var = 42;
auto lambda_func = [&int_var]() { cout <<
  "This lambda captures int_var by reference: " << int_var << endl;};
for (int i = 0; i<3; i++) {
  int_var ++;
  lambda_func();
}
```

Another example:

```
int sum = 0;
for (int i=0; i<3; i++)
  [&sum](int x){sum += x;}(i);
```

STL and lambda:

```
class Thing {
public:
  //various public member functions
  void save(ostream& os) const; // write our data to the provided
                                // output stream.
private:
  // member variables
};
```

Let's create a bunch of Things objects with new, and saved pointers in

```
list<Thing *> thing_ptrs;
```

Let's tell each Thing in the list to write its output to a file stream object named out_file using Thing's save function.
First, we'll open the file and then use an explicit for loop to get the iterator pointing to each Thing pointer:

```
ofstream out_file("output.txt"); // open a file for output.
for(auto it = thing_ptrs.begin(); it != thing_ptrs.end(); ++it)
  (*it) -> save(out_file);
```

We can use lambda instead:

```
for_each(thing_ptrs.begin(), thing_ptrs.end(),
  [&out_file](Thing* ptr){ ptr->save(out_file);});
```

Accessing member variables in a lambda that is in a member function:

```
class Gizmo {
public:
  int get_count_over_criterion();

private:
  std::list<int> ints;
  int criterion;
  int count;
};
```

The this pointer is a local variable so you can capture it, and capturing by value is fine since it can't be modified any way. Then in the body of lambda. the compiler knows that count and criterion are member variables because it has seen the class declaration, and rewrites the references to them using the this pointer which thanks to the capture, it knows about within the lambda body.

```
int Gizmo::get_count_over_criterion()
{
  count = 0;
  for_each(ints.begin(), ints.end(),
    [this](int i){ if (i > criterion) count++;});
  return count;
}
```

both are class memeber variables captured by this.