Lambda 表达式是 C++11 中最重要的新特性之一，而 Lambda 表达式，实际上就是提供了一个类似匿名函数的特性，而匿名函数则是在需要一个函数，但是又不想费力去命名一个函数的情况下去使用的。这样的场景其实有很多很多，所以匿名函数几乎是现代编程语言的标配。

Lambda 表达式基础

Lambda 表达式的基本语法如下：

```
[捕获列表](参数列表) mutable(可选) 异常属性 -> 返回类型 {
 // 函数体
}
```

上面的语法规则除了［捕获列表］内的东西外，其他部分都很好理解，只是一般函数的函数名被略去，返回值使用了一个 -> 的形式进行（我们在上一节前面的尾返回类型已经提到过这种写法了）。

所谓捕获列表，其实可以理解为参数的一种类型，lambda 表达式内部函数体在默认情况下是不能够使用函数体外部的变量的，这时候捕获列表可以起到传递外部数据的作用。根据传递的行为，捕获列表也分为以下几种：

1. 值捕获

与参数传值类似，值捕获的前期是变量可以拷贝，不同之处则在于，被捕获的变量在 lambda 表达式被创建时拷贝，而非调用时才拷贝：

```
void learn_lambda_func_1() {
 int value_1 = 1;
 auto copy_value_1 = [value_1] {
 return value_1;
 };
 value_1 = 100;
 auto stored_value_1 = copy_value_1();
 // 这时, stored_value_1 == 1, 而 value_1 == 100.
 // 因为 copy_value_1 在创建时就保存了一份 value_1 的拷贝
  cout << "value_1 = " << value_1 << endl;
  cout << "stored_value_1 = " << stored_value_1 << endl;
}
```

2. 引用捕获

与引用传参类似，引用捕获保存的是引用，值会发生变化。

```
void learn_lambda_func_2() {
 int value_2 = 1;
 auto copy_value_2 = [&value_2] {
 return value_2;
 };
```

```
value_2 = 100;
auto stored_value_2 = copy_value_2();
// 这时, stored_value_2 == 100, value_1 == 100.
// 因为 copy_value_2 保存的是引用
 cout << "value_2 = " << value_2 << endl;
 cout << "stored_value_2 = " << stored_value_2 << endl;
}
```

3. 隐式捕获

手动书写捕获列表有时候是非常复杂的，这种机械性的工作可以交给编译器来处理，这时候可以在捕获列表中写一个 `&` 或 `=` 向编译器声明采用 引用捕获或者值捕获.

总结一下，捕获提供了lambda 表达式对外部值进行使用的功能，捕获列表的最常用的四种形式可以是：

- [] 空捕获列表
- [name1, name2, ...] 捕获一系列变量
- [&] 引用捕获, 让编译器自行推导捕获列表
- [=] 值捕获, 让编译器执行推导应用列表

4. 表达式捕获(C++14)

> 这部分内容需要了解后面马上要提到的右值引用以及智能指针

上面提到的值捕获、引用捕获都是已经在外层作用域声明的变量，因此这些捕获方式捕获的均为左值，而不能捕获右值。

C++14 给与了我们方便，允许捕获的成员用任意的表达式进行初始化，这就允许了右值的捕获，被声明的捕获变量类型会根据表达式进行判断，判断方式与使用 `auto` 本质上是相同的：

```
#include <iostream>
#include <utility>
void learn_lambda_func_3(){
 auto important = std::make_unique<int>(1);
 auto add = [v1 = 1, v2 = std::move(important)](int x, int y) -> int {
 return x+y+v1+(*v2);
 };
 std::cout << "add(3, 4) = " << add(3, 4) << std::endl;
}
```

在上面的代码中，`important` 是一个独占指针，是不能够被捕获到的，这时候我们需要将其转移为右值，在表达式中初始化。

泛型 Lambda (C++14)

上一节中我们提到了 `auto` 关键字不能够用在参数表里，这是因为这样的写法会与模板的功能产生冲突。但是 Lambda 表达式并不是普通函数，所以 Lambda

表达式并不能够模板化。这就为我们造成了一定程度上的麻烦：参数表不能够泛化，必须明确参数表类型。
幸运的是，这种麻烦只存在于 C++11 中，从 C++14 开始，Lambda 函数的形式参数可以使用 `auto` 关键字来产生意义上的泛型：

```
void learn_lambda_func_4(){
 auto generic = [](auto x, auto y) {
 return x+y;
 };

 std::cout << "generic(1,2) = " << generic(1, 2) << std::endl;
 std::cout << "generic(1.1,2.2) = " << generic(1.1, 2.2) << std::endl;
}

```