## C++ 11 Lambda表达式

C++11的一大亮点就是引入了Lambda表达式。利用Lambda表达式，可以方便的定义和创建匿名函数。对于C++这门语言来说来说，“Lambda表达式”或“匿名函数”这些概念听起来好像很深奥，但很多高级语言在很早以前就已经提供了Lambda表达式的功能，如C#，Python等。今天，我们就来简单介绍一下C++中Lambda表达式的简单使用。

## 声明Lambda表达式

Lambda表达式完整的声明格式如下：

```
[capture list] (params list) mutable exception-> return type { function body }
```

各项具体含义如下

1. capture list：捕获外部变量列表
2. params list：形参列表
3. mutable指示符：用来说用是否可以修改捕获的变量
4. exception：异常设定
5. return type：返回类型
6. function body：函数体

此外，我们还可以省略其中的某些成分来声明“不完整”的Lambda表达式，常见的有以下几种：

| 序号 | 格式 |
| --- | --- |
| 1 | [capture list] (params list) -> return type {function body} |
| 2 | [capture list] (params list) {function body} |
| 3 | [capture list] {function body} |

其中：

- 格式1声明了const类型的表达式，这种类型的表达式不能修改捕获列表中的值。
- 格式2省略了返回值类型，但编译器可以根据以下规则推断出Lambda表达式的返回类型：（1）：如果function body中存在return语句，则该Lambda表达式的返回类型由return语句的返回类型确定；（2）：如果function body中没有return语句，则返回值为void类型。
- 格式3中省略了参数列表，类似普通函数中的无参函数。

讲了这么多，我们还没有看到Lambda表达式的庐山真面目，下面我们就举一个实例。

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

bool cmp(int a, int b)
{
    return  a < b;
}

int main()
{
    vector<int> myvec{ 3, 2, 5, 7, 3, 2 };
    vector<int> lbvec(myvec);

    sort(myvec.begin(), myvec.end(), cmp); // 旧式做法
    cout << "predicate function:" << endl;
    for (int it : myvec)
        cout << it << ' ';
    cout << endl;

    sort(lbvec.begin(), lbvec.end(), [](int a, int b) -> bool { return a < b; });   // Lambda表达式
    cout << "lambda expression:" << endl;
    for (int it : lbvec)
        cout << it << ' ';
}
```

在C++11之前，我们使用STL的sort函数，需要提供一个谓词函数。如果使用C++11的Lambda表达式，我们只需要传入一个匿名函数即可，方便简洁，而且代码的可读性也比旧式的做法好多了。

下面，我们就重点介绍一下Lambda表达式各项的具体用法。

## 捕获外部变量

Lambda表达式可以使用其可见范围内的外部变量，但必须明确声明（明确声明哪些外部变量可以被该Lambda表达式使用）。那么，在哪里指定这些外部变量呢？Lambda表达式通过在最前面的方括号[]来明确指明其内部可以访问的外部变量，这一过程也称过Lambda表达式“捕获”了外部变量。

我们通过一个例子来直观地说明一下：

```
#include <iostream>
using namespace std;

int main()
{
    int a = 123;
    auto f = [a] { cout << a << endl; };
    f(); // 输出：123

    //或通过“函数体”后面的‘()’传入参数
    auto x = [](int a){cout << a << endl;}(123);
}
```

上面这个例子先声明了一个整型变量a，然后再创建Lambda表达式，该表达式“捕获”了a变量，这样在Lambda表达式函数体中就可以获得该变量的值。

类似参数传递方式（值传递、引入传递、指针传递），在Lambda表达式中，外部变量的捕获方式也有值捕获、引用捕获、隐式捕获。

**1、值捕获**

值捕获和参数传递中的值传递类似，被捕获的变量的值在Lambda表达式创建时通过值拷贝的方式传入，因此随后对该变量的修改不会影响影响Lambda表达式中的值。

示例如下：

```
int main()
{
    int a = 123;
    auto f = [a] { cout << a << endl; };
    a = 321;
    f(); // 输出：123
}
```

这里需要注意的是，如果以传值方式捕获外部变量，则在Lambda表达式函数体中不能修改该外部变量的值。

**2、引用捕获**

使用引用捕获一个外部变量，只需要在捕获列表变量前面加上一个引用说明符&。如下：

```
int main()
{
    int a = 123;
    auto f = [&a] { cout << a << endl; };
    a = 321;
    f(); // 输出：321
}
```

从示例中可以看出，引用捕获的变量使用的实际上就是该引用所绑定的对象。

**3、隐式捕获**

上面的值捕获和引用捕获都需要我们在捕获列表中显示列出Lambda表达式中使用的外部变量。除此之外，我们还可以让编译器根据函数体中的代码来推断需要捕获哪些变量，这种方式称之为隐式捕获。隐式捕获有两种方式，分别是[=]和[&]。[=]表示以值捕获的方式捕获外部变量，[&]表示以引用捕获的方式捕获外部变量。

隐式值捕获示例：

```
int main()
{
    int a = 123;
    auto f = [=] { cout << a << endl; };    // 值捕获
    f(); // 输出：123
}
```

隐式引用捕获示例：

```
int main()
{
    int a = 123;
    auto f = [&] { cout << a << endl; };    // 引用捕获
    a = 321;
    f(); // 输出：321
}
```

**4、混合方式**

上面的例子，要么是值捕获，要么是引用捕获，Lambda表达式还支持混合的方式捕获外部变量，这种方式主要是以上几种捕获方式的组合使用。

到这里，我们来总结一下：C++11中的Lambda表达式捕获外部变量主要有以下形式：

| 捕获形式 | 说明 |
| --- | --- |
| [] | 不捕获任何外部变量 |
| [变量名, …] | 默认以值得形式捕获指定的多个外部变量（用逗号分隔），如果引用捕获，需要显示声明（使用&说明符） |
| [this] | 以值的形式捕获this指针 |
| [=] | 以值的形式捕获所有外部变量 |
| [&] | 以引用形式捕获所有外部变量 |
| [=, &x] | 变量x以引用形式捕获，其余变量以传值形式捕获 |
| [&, x] | 变量x以值的形式捕获，其余变量以引用形式捕获 |

## 修改捕获变量

前面我们提到过，在Lambda表达式中，如果以传值方式捕获外部变量，则函数体中不能修改该外部变量，否则会引发编译错误。那么有没有办法可以修改值捕获的外部变量呢？这是就需要使用mutable关键字，该关键字用以说明表达式体内的代码可以修改值捕获的变量，示例：

```
int main()
{
    int a = 123;
    auto f = [a]()mutable { cout << ++a; }; // 不会报错
    cout << a << endl; // 输出：123
    f(); // 输出：124
}
```

## Lambda表达式的参数

Lambda表达式的参数和普通函数的参数类似，那么这里为什么还要拿出来说一下呢？原因是在Lambda表达式中传递参数还有一些限制，主要有以下几点：

1. 参数列表中不能有默认参数
2. 不支持可变参数
3. 所有参数必须有参数名

常用举例：

```
  {
      int m = [](int x) { return [](int y) { return y * 2; }(x)+6; }(5);
      std::cout << "m:" << m << std::endl;               //输出m:16

      std::cout << "n:" << [](int x, int y) { return x + y; }(5, 4) << std::endl;            //输出n:9

      auto gFunc = [](int x) -> function<int(int)> { return [=](int y) { return x + y; }; };
      auto lFunc = gFunc(4);
      std::cout << lFunc(5) << std::endl;

      auto hFunc = [](const function<int(int)>& f, int z) { return f(z) + 1; };
      auto a = hFunc(gFunc(7), 8);

      int a = 111, b = 222;
      auto func = [=, &b]()mutable { a = 22; b = 333; std::cout << "a:" << a << " b:" << b << std::endl; };

      func();
      std::cout << "a:" << a << " b:" << b << std::endl;

      a = 333;
      auto func2 = [=, &a] { a = 444; std::cout << "a:" << a << " b:" << b << std::endl; };
      func2();

      auto func3 = [](int x) ->function<int(int)> { return [=](int y) { return x + y; }; };


      std::function<void(int x)> f_display_42 = [](int x) { print_num(x); };
      f_display_42(44);
  }
```