C++ Lambda Story

本文主要阐述一个现代C++的重要特性——Lambda表达式，从C++ 03开始讨论直至C++20，具体安排如下：

- C++03： 该部分主要引入为什么需要Lambda表达式
- C++11： 在C++ 11标准中已经囊括了关于Lambda表达式的绝大部分元素以及一些使用技巧
- C++14： 关于Lambda的一些特性更新
- C++17： 关于this指针以及constexpr的改进
- C++20： 该部分仅仅简要概述部分更新的Lambda表达式特性

一、C++03中的Lambda

在早期的STL库中，可以在任何的容器中调用`std::sort`对容器内的元素进行排序，但是在C++03中仅能使用function pointers以及functors，如下例所示：

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

struct PrintFunctor {
    void operator()(int x) const {
        std::cout << x << std::endl;
    }
};

int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    std::for_each(v.begin(), v.end(), PrintFunctor());
}
```

该例使用operator()定义了一个简单的functor。

尽管使用function pointer也可以做到上述的打印效果，但是functor可以做更多的事以及附带一些状态，例如对调用次数进行计数：

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

struct PrintFunctor {
    PrintFunctor(): numCalls(0) { }

    void operator()(int x) const {
        std::cout << x << std::endl;
        ++numCalls;
    }

    mutable int numCalls;
};
```

```
int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    PrintFunctor visitor = std::for_each(v.begin(), v.end(), PrintFunctor());
    std::cout << "num calls:" << visitor.numCalls << std::endl;
};
```

该例中使用了numCalls来统计operator的调用次数。代码中需注意的是，虽然operator()是一个const类型的成员函数，但是使用了mutable修饰的numCalls是可修改的。

除此之外，我们还可以使用functor获取调用域内的变量，做法很简单，讲捕获到的变量赋给struct内的一个变量即可，具体做法如下所示：

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

struct PrintFunctor {
    PrintFunctor(const std::string& str):
        strVar(str), numCalls(0) { }

    void operator()(int x) const {
        std::cout << strVar << x << std::endl;
        ++numCalls;
    }

    std::string strVar;
    mutable int numCalls;
};

int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    const std::string introStr("Elem: ");
    PrintFunctor visitor = std::for_each(v.begin(), v.end(),
PrintFunctor(introStr));
    std::cout << "num calls:" << visitor.numCalls << std::endl;
}
```

可以看到在该例中，functor捕获了一个外部的introStr变量，并将之用于call operator中。

从上面的例子中可以看出functor非常强大，它可以替换function pointer的同时携带额外的状态信息，也可以捕获调用域范围内的任意变量。我们可以以任意偏好的方式设计它。但是它的不足之处也非常明显，即我们需要额外的写出一个独立的类，可以是在同一个cpp文件下，也可以在不同的cpp的文件下。但在C++03中是不可将之写在一个function内，如下所示：

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

int main() {
    struct PrintFunctor {
```

```
        void operator()(int x) const {
            std::cout << x << std::endl;
        }
    };

    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    std::for_each(v.begin(), v.end(), PrintFunctor());
}
```

上述代码使用-std=c++98的clang编译会得到如下错误信息：

```
a_local_functor.cpp:15:5: warning: template argument uses local type
      'PrintFunctor' [-Wlocal-type-template-args]
    std::for_each(v.begin(), v.end(), PrintFunctor());
```

即在C++98/03中不可使用local type实例化一个template。但在C++11中，委员会解除了使用local type实例化一个template的限制，即上述代码在C++11以后的编译器中可通过编译并运行，也就意味着你可以在靠近你要使用的地方定义functor。

但是C++11也带来了另外一个思路：即让编译器帮我们"写"一个这样的functor。这也就意味着需要新的语法更简洁的为我们“写”一个functor——这也就是Lambda表达式诞生的缘由。

在下一章节我们便初步探讨Lambda表达式。

二、C++11中的Lambda

自从C++委员会引用Lambda以来，它迅速的成为了现代C++中最具辨识度的特性之一。有兴趣的读者可以查阅C++11最终草案N33371以及Lambda单独章节。

Lambda以一种非常聪明的方式加入语言特性中——虽然使用了新语法，但是编译器实际却是将它扩展成了一个真实的class。这样以来，C++就拥有了强类型语言的所有优点（当然缺点不可避免）。

本章主要从以下几点切入：

- lambda基本语法
- 如何捕捉变量
- 如何捕捉成员变量
- lambda的返回类型
- 什么是闭包
- 一些边界条件
- lambda转化为function pointer
- IIFE

2.1 Lambda语法

下面给出一个基础的Lambda和一个对应的local functor：

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

int main() {
    struct {
        void operator()(int x) const{
```

```
            std::cout << x << std::endl;
        }
    } aLocalFunctor;

    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    std::for_each(v.begin(), v.end(), aLocalFunctor);
    std::for_each(v.begin(), v.end(), [](int x) {
            std::cout << x << std::endl;
            }
    );
}
```

上述代码中，编译器将：

```
[](int x) {std::cout << x << std::endl;}
```

转化为类似下面的local type class：

```
struct {
    void operator()(int x) const {
        std::cout << x << std::endl;
    }
} someInstance;
```

Lambda表达式的基本语法如下所示：

```
[] ()   { code; }
^  ^  ^
|  |  |
|  |  optional: mutable, exception, trailing return, ...
|  |
|  optional: parameter list
|
lambda introducer with capture list
```

expr.prim.lambda#2中写到：

> 调用Lambda后会返回一个临时的prvalue（纯右值）叫做“闭包”的object

expr.prim.lambda#3中写到：

> Lambda表达式是一个独一无二的未命名的非union类型的class type，也叫闭包type

Lambda表达式的一些示例如下：

```
[]{} // the simplest lambda
[](float f, int a) { return a * f;}
[](MyClass t) -> int { auto a = t.compute(); return a;}
[](int a, int b) { return a < b; }
[x](int a, int b) mutable { return a < b; ++x; }
```

2.2 Lambda的type

由于编译器会为每个lambda生成一个唯一的名字，因此无法预先知道它。这也是为什么我们必须使用auto（或decltype）来推断它的类型。

```
auto myLambda = [](int a) -> double { return 2.0 * a; }
```

expr.prim.lambda中写到

> lambda表达式的闭包类型拥有一个deleted default constructor和一个deleted copy assignment operator。它具有一个隐式声明的copy constructor，并可能具有一个隐式声明的move constructor。

因此如果这么写：

```
auto foo = [&x, &y]() { ++x; ++y; };
decltype(foo) fooCopy;
```

使用gcc编译时会报如下错误：

```
error: use of deleted function ‘main()::<lambda()>::<lambda>()’
     decltype(foo) fooCopy;
                   ^~~~~~
note: a lambda closure type has a deleted default constructor
     auto foo = [](){};
```

另一方面需要注意的是，如果有两个函数体完全相同的lambda：

```
auto firstLam = [](int x) { return x * 2; };
auto secondLam = [](int x) { return x * 2; };
static_assert(std::is_same<decltype(firstLam), decltype(secondLam)>::value, "not
same");
```

编译时会报如下错误：

```
error: static assertion failed: not same
     static_assert(std::is_same<decltype(firstLam), decltype(secondLam)>::value,
"not same");
     ^~~~~~~~~~~~~
```

但是如果copy lambdas：

```
auto firstLam = [](int x) { return x * 2; };
auto secondLam = firstLam;
static_assert(std::is_same<decltype(firstLam), decltype(secondLam)>::value, "not
same");
```

则可以正常编译，复制lambda的同时还会复制其状态，这一点后面会继续讲到。

2.3 capture list

[]不仅引入了lambda，而且还包含捕获变量的列表。称为“capture list”。

通过捕获变量，我们可以在闭包类型中创建该变量的成员**副本**。然后，在lambda主体内部，可以对其进行访问。

在C ++ 03章节中，我们为PrintFunctor做了类似的事情。在该类中，我们添加了一个成员变量string strText;。这是在构造函数中初始化的。

capture list的基本语法如下：

- [&] - 按引用捕获，通过引用捕获可达范围内声明的所有变量
- [=] - 按值捕获，将复制一个值
- [x, &y] - 按值捕获x，按引用捕获y

比如：

```
std::string str {"Hello World"};
auto foo = [str]() { std::cout << str << '\n'; };
foo();
```

对于上述lambda，编译器可能会生成以下local functor：

```
struct _unnamedLambda {
    _unnamedLambda(std::string s) : str(s) { }
    void operator() const {
        std::cout << str << '\n';
    }

    std::string str;
};
```

变量被传递到构造函数中，该构造函数在lambda声明中被调用。

确切地说，标准expr.prim.lambda#21中提到：

> 评估lambda表达式时，将使用通过copy捕获的automatic storage duration值初始化生成的闭包对象中对应的非静态数据成员。

上面显示的一个可能的构造函数（_unnamedLambda）仅用于演示，因为编译器可能会以不同的方式实现它，并且不会公开它。

```
int x = 1, y = 1;
std::cout << x << " " << y << std::endl;
auto foo = [&x, &y]() { ++x; ++y; };
foo();
std::cout << x << " " << y << std::endl;
```

对于上述lambda，编译器可能会生成以下local functor：

```
struct _unnamedLambda {
    _unnamedLambda(int& a, int& b) : x(a), y(b) { }
    void operator() const {
        ++x; ++y;
    }

    int& x;
    int& y;
};
```

由于我们通过引用捕获x和y，因此闭包类型将包含同样是引用的成员变量。

虽然指定[=]或[&]可能很方便，但由于它会捕获所有自动存储持续时间变量，所以更明确地捕获一个变量会更方便。您还可以在Scott Meyers的“ Effective Modern C ++”的第31项中阅读更多内容：“避免使用默认捕获模式。”

2.4 Mutable

默认情况下，闭包类型的operator（）是const，您不能在lambda主体内修改捕获的变量。

如果要更改此行为，则需要在参数列表之后添加mutable关键字：

```
int x = 1, y = 1;
std::cout << x << " " << y << std::endl;   // 1 1
auto foo = [x, y]() mutable { ++x; ++y; };
foo();
std::cout << x << " " << y << std::endl;  // 1 1
```

在上面的示例中，我们可以更改x和y的值。当然，由于这些只是封闭范围中x和y的副本，因此在调用foo之后我们看不到它们的新值。另一方面，如果您通过引用捕获，则在非可变lambda中，您无法重新绑定引用，但是可以更改引用的变量。

```
int x = 1;
std::cout << x << std::endl;   // 1
auto foo = [&x]() mutable { ++x; };
foo();
std::cout << x << std::endl;  // 2
```

2.5 Capturing Globals

如果您具有全局值，然后在lambda中使用[=]，则您可能会认为全局值也可以被捕获，但是事实并非如此。

```
#include <iostream>
int global = 10;

int main()
{
    std::cout << global << std::endl;
    auto foo = [=] () mutable { ++global; };
    foo();
    std::cout << global << std::endl;
    [] { ++global; } ();
    std::cout << global << std::endl;
    [global] { ++global; } ();
}
```

Lambda仅捕获具有自动存储持续时间的变量。 g++编译时会报如下错误：

```
warning: capture of variable ‘global’ with non-automatic storage duration
```

仅当您明确捕获全局变量时才会显示此警告，因此，如果使用[=]，编译器将不会报错。

clang++编译器给出的提示信息更加明确：

```
error: 'global' cannot be captured because it does not have
      automatic storage duration
```

2.6 Capturing Statics

与捕获全局变量类似，您将获得与静态变量相同的结果：

```
#include <iostream>

void bar()
{
    static int static_int = 10;
    std::cout << static_int << std::endl;
    auto foo = [=] () mutable { ++static_int; };
    foo();
    std::cout << static_int << std::endl;
    [] { ++static_int; } ();
    std::cout << static_int << std::endl;
    [static_int] { ++static_int; } ();
}

int main()
{
    bar();
}
```

输出如下：

```
10
11
12
```

同样，对于bar中最后一行代码，仅当您显式捕获静态变量时才会出现警告，如果使用[=]，则不会出现警告。

2.7 Capturing a Class Member And this

在类方法中，事情变得更加复杂：

```
#include <iostream>

struct Baz {
    void foo() {
        auto lam = [s]() { std::cout << s; };
        lam();
    }

    std::string s;
};

int main() {
    Baz b;
    b.foo();
}
```

该代码尝试捕获s，它是成员变量。但是编译器将发出错误消息：

```
error: capture of non-variable ‘Baz::s’
error: ‘this’ was not captured for this lambda function
```

要解决此问题，您必须捕获this指针。然后，您将可以访问成员变量。

我们可以将代码更新为：

```
struct Baz {
    void foo() {
        auto lam = [this]() { std::cout << s; };
        lam();
    }
    std::string s;
};
```

现在没有生成任何编译器错误。

您也可以使用[=]或[＆]来捕获它（它们都具有相同的效果！），但是请注意，我们是通过值捕获到指针的。因此，您可以访问成员变量，而不是其副本。在C ++ 11（甚至在C ++ 14）中，您不能编写如下代码捕获对象的副本：

```
auto lam = [*this]() { std::cout << s; };
```

如果您在单一方法的上下文中使用lambda，则可以捕获它。但是更复杂的情况呢？您知道以下代码会发生什么吗？

```
#include <iostream>
#include <functional>

struct Baz
{
    std::function<void()> foo()
    {
        return [=] { std::cout << s << std::endl; };
    }

    std::string s;
};

int main()
{
    auto f1 = Baz{"ala"}.foo();
    auto f2 = Baz{"ula"}.foo();
    f1();
    f2();
}
```

该代码声明一个Baz对象，然后调用foo()。请注意，foo()返回一个lambda（存储在std :: function中），该lambda捕获该类的成员。

由于我们使用**临时对象**，因此我们无法确定调用f1和f2时会发生什么。这是一个悬而未决的参考问题，会产生未定义的行为。

类似于：

```
struct Bar {
    std::string const& foo() const { return s; };
    std::string s;
};
auto&& f1 = Bar{"ala"}.foo(); // dangling reference
```

同样，如果您明确声明捕获（[s]）：

```
std::function<void()> foo()
{
    return [s] { std::cout << s << std::endl; };
}
```

总而言之，当lambda可以超过它本身的作用域"存活"时，捕获this可能会变得棘手。当您使用异步调用或多线程时，可能会发生这种情况。

我们将在C ++ 17章中返回该主题。

2.8 Move-able-only Objects

如果您有一个只能移动的对象（例如unique_ptr），则不能将其作为捕获变量移动到lambda。按值进行捕获是行不通的，因此您只能通过引用进行捕获...但是，这不会转移所有权，而且可能不是您想要的。

```
std::unique_ptr<int> p(new int{10});
auto foo = [p] () {}; // does not compile...
```

2.9 Preserving Const

如果捕获const变量，则将保留常亮性：

```
int const x = 10;
auto foo = [x] () mutable {
    std::cout << std::is_const<decltype(x)>::value << std::endl;
    x = 11;
};
foo();
```

2.10 Return Type

在C ++ 11中，您可以跳过lambda的尾随返回类型，然后编译器将为您推断出该类型。

最初，返回类型推导仅限于带有包含单个return语句的主体的lambda，但由于实施更方便的版本没有问题，因此这一限制很快被取消。

因此，从C ++ 11开始，只要所有的return语句都属于同一类型，编译器就可以推断出返回类型，如：

```
auto baz = [] () {
    int x = 10;
    if ( x < 20)
        return x * 1.1;
    else
        return x * 2.1;
};
```

在上面的lambda中，我们有两个return语句，但是它们都指向double，因此编译器可以推断出类型。

在C ++ 14中，lambda的返回类型将被更新以适应常规函数的自动类型推导规则。

2.11 IIFE - Immediately Invoked Function Expression

在我们的示例中，我定义了一个lambda，然后使用闭包对象对其进行了调用……但是您也可以立即调用它：

```
int x = 1, y = 1;
[&]() { ++x; ++y; }(); // <-- call ()
std::cout << x << " " << y << std::endl;
```

当您对const对象进行复杂的初始化时，此类表达式可能会很有用。

```
const auto val = []() { /* several lines of code... */ }();
```

2.12 Conversion to a Function Pointer

没有lambda-capture的lambda表达式的闭包类型具有指向该函数的指针的non-virtual non-explicit const conversion function，该函数具有与闭包类型的函数调用运算符相同的参数和返回类型。此转换函数返回的值应为一个函数的地址，该函数在被调用时与调用闭包类型的函数调用运算符具有相同的作用。

换句话说，您可以将没有lambda-capture的lambda转换为函数指针。

比如：

```
#include <iostream>

void callWith10(void(* bar)(int))
{
    bar(10);
}

int main()
{
    struct
    {
        using f_ptr = void(*)(int);
        void operator()(int s) const { return call(s); }
        operator f_ptr() const { return &call; }

    private:
        static void call(int s) { std::cout << s << std::endl; };
    } baz;

    callWith10(baz);
    callWith10([](int x) { std::cout << x << std::endl; });
}
```

2.13 C++11中的Lambda总结

在本章中，主要包括了如何创建和使用lambda表达式。本章描述了语法，捕获相关规则，lambda的type等。

Lambda表达式成为现代C ++的重要标志之一。通过更多用例，开发人员还看到了改进lambda的可能性。这就是为什么您现在可以进入下一章并查看委员会在C ++ 14中添加的更新的原因。

三、C++14中的Lambda

四、C++17中的Lambda

五、 C++20中的Lambda

> Written with StackEdit.