

## C++ 中的 Lambda 表达式

# C++ 中的 Lambda 表达式

Visual Studio 2015 [其他版本](#) ▾

在 C++ 11 中，lambda 表达式（通常称为 "lambda"）是一种在被调用的位置或作为参数传递给函数的位置定义匿名函数对象的简便方法。Lambda 通常用于封装传递给算法或异步方法的少量代码行。本文定义了 lambda 是什么，将 lambda 与其他编程技术进行比较，描述其优点，并提供一个基本示例。

## Lambda 表达式的各部分

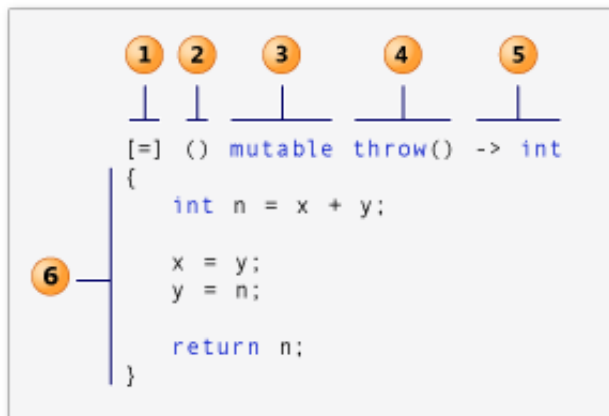
ISO C++ 标准展示了作为第三个参数传递给 **std::sort()** 函数的简单 lambda：

C++

```
#include <algorithm>
#include <cmath>

void abssort(float* x, unsigned n) {
    std::sort(x, x + n,
        // Lambda expression begins
        [](float a, float b) {
            return (std::abs(a) < std::abs(b));
        } // end of lambda expression
    );
}
```

此图显示了 lambda 的组成部分：



1. Capture 子句 ( 在 C++ 规范中也称为 lambda 引导。 )
2. 参数列表 ( 可选 )。 ( 也称为 lambda 声明符 )
3. 可变规范 ( 可选 )。
4. 异常规范 ( 可选 )。
5. 尾随返回类型 ( 可选 )。
6. “lambda 体”



## Capture 子句

Lambda 可在其主体中引入新的变量 ( 用 **C++14** )，它还可以访问 ( 或“捕获”) 周边范围内的变量。Lambda 以 Capture 子句 ( 标准语法中的 lambda 引导 ) 开头，它指定要捕获的变量以及是通过值还是引用进行捕获。有与号 (&) 前缀的变量通过引用访问，没有该前缀的变量通过值访问。

空 capture 子句 `[]` 指示 lambda 表达式的主体不访问封闭范围中的变量。

可以使用默认捕获模式 ( 标准语法中的 `capture-default` ) 来指示如何捕获 lambda 中引用的任何外部变量：`[&]` 表示通过引用捕获引用的所有变量，而 `[=]` 表示通过值捕获它们。可以使用默认捕获模式，然后为特定变量显式指定相反的模式。例如，如果 lambda 体通过引用访问外部变量 `total` 并通过值访问外部变量 `factor`，则以下 capture 子句等效：

**C++**

```
[&total, factor]
[factor, &total]
[&, factor]
[factor, &]
[=, &total]
[&total, =]
```

使用 `capture-default` 时，只有 lambda 中提及的变量才会被捕获。

如果 capture 子句包含 `capture-default&`，则该 capture 子句的 identifier 中没有任何 capture 可采用

& identifier 形式。同样，如果 capture 子句包含 capture-default=，则该 capture 子句的 capture 不能采用 = identifier 形式。identifier 或 **this** 在 capture 子句中出现的次数不能超过一次。以下代码片段给出了一些示例。

C++

```
struct S { void f(int i); };

void S::f(int i) {
    [&, i]{};    // OK
    [&, &i]{};   // ERROR: i preceded by & when & is the default
    [=, this]{}; // ERROR: this when = is the default
    [i, i]{};    // ERROR: i repeated
}
```

capture 后跟省略号是包扩展，如以下[可变参数模板](#)示例中所示：

C++

```
template<class... Args>
void f(Args... args) {
    auto x = [args...] { return g(args...); };
    x();
}
```

要在类方法的正文中使用 lambda 表达式，请将 **this** 指针传递给 Capture 子句，以提供对封闭类的方法和数据成员的访问权限。有关展示如何将 lambda 表达式与类方法一起使用的示例，请参阅[Lambda 表达式的示例](#)中的“示例：在方法中使用 Lambda 表达式”。

在使用 capture 子句时，建议你记住以下几点（尤其是使用采取多线程的 lambda 时）：

- 引用捕获可用于修改外部变量，而值捕获却不能实现此操作。（**mutable**允许修改副本，而不能修改原始项。）
- 引用捕获会反映外部变量的更新，而值捕获不会反映。
- 引用捕获引入生存期依赖项，而值捕获却没有生存期依赖项。当 lambda 以异步方式运行时，这一点尤其重要。如果在异步 lambda 中通过引用捕获本地变量，该本地变量将很可能在 lambda 运行时消失，从而导致运行时访问冲突。

## 通用捕获 (C++14)

在 C++14 中，可在 Capture 子句中引入并初始化新的变量，而无需使这些变量存在于 lambda 函数的封闭范围内。初始化可以任何任意表达式表示；且将从该表达式生成的类型推导新变量的类型。此功能的一个好处是，在 C++14 中，可从周边范围捕获只移动的变量（例如 `std::unique_ptr`）并在 lambda 中使用它们。

```
pNums = make_unique<vector<int>>(nums);
//...
    auto a = [ptr = move(pNums)]()
    {
        // use ptr
    };
```



## 参数列表

除了捕获变量，lambda 还可接受输入参数。参数列表（在标准语法中称为 lambda 声明符）是可选的，它在大多数方面类似于函数的参数列表。

```
int y = [] (int first, int second)
{
    return first + second;
};
```

在 **C++14** 中，如果参数类型是泛型，则可以使用 auto 关键字作为类型说明符。这将告知编译器将函数调用运算符创建为模板。参数列表中的每个 auto 实例等效于一个不同的类型参数。

```
auto y = [] (auto first, auto second)
{
    return first + second;
};
```

lambda 表达式可以将另一个 lambda 表达式作为其参数。有关详细信息，请参阅 [Lambda 表达式的示例](#) 主题中的“高阶 Lambda 表达式”。

由于参数列表是可选的，因此在不将参数传递到 lambda 表达式，并且其 lambda-declarator: 不包含 exception-specification、trailing-return-type 或 **mutable** 的情况下，可以省略空括号。



## 可变规范

通常，lambda 的函数调用运算符为 const-by-value，但对 **mutable** 关键字的使用可将其取消。它不会生成可变的数据成员。利用可变规范，lambda 表达式的主体可以修改通过值捕获的变量。本文后面的一些示例将显示如何使用 **mutable**。



## 异常规范

你可以使用 **throw()** 异常规范来指示 lambda 表达式不会引发任何异常。与普通函数一样，如果 lambda 表达式声明 [C4297](#) 异常规范且 lambda 体引发异常，Visual C++ 编译器将生成警告 **throw()**，如下所示：

C++

```
// throw_lambda_expression.cpp
// compile with: /W4 /EHsc
int main() // C4297 expected
{
    []() throw() { throw 5; }();
}
```

有关详细信息，请参阅[异常规范 \(throw\) \(C++\)](#)。



## 返回类型

将自动推导 lambda 表达式的返回类型。无需使用 **auto** 关键字，除非指定尾随返回类型。trailing-return-type 类似于普通方法或函数的返回类型部分。但是，返回类型必须跟在参数列表的后面，你必须在返回类型前面包含 trailing-return-type 关键字 **->**。

如果 lambda 体仅包含一个返回语句或其表达式不返回值，则可以省略 lambda 表达式的返回类型部分。如果 lambda 体包含单个返回语句，编译器将从返回表达式的类型推导返回类型。否则，编译器会将返回类型推导为 **void**。下面的代码示例片段说明了这一原则。

C++

```
auto x1 = [](int i){ return i; }; // OK: return type is int
auto x2 = []{ return{ 1, 2 }; }; // ERROR: return type is void, deducing
                                // return type from braced-init-list is not valid
```

lambda 表达式可以生成另一个 lambda 表达式作为其返回值。有关详细信息，请参阅 [Lambda 表达式的示例](#) 中的“高阶 Lambda 表达式”。



## Lambda 体

lambda 表达式的 lambda 体（标准语法中的 compound-statement）可包含普通方法或函数的主体可包含的任何内容。普通函数和 lambda 表达式的主体均可访问以下变量类型：

- 从封闭范围捕获变量，如前所述。
- 参数

- 本地声明变量
- 类数据成员（在类内部声明并且捕获 **this** 时）
- 具有静态存储持续时间的任何变量（例如，全局变量）

以下示例包含通过值显式捕获变量 **n** 并通过引用隐式捕获变量 **m** 的 lambda 表达式：

**C++**

```
// captures_lambda_expression.cpp
// compile with: /W4 /EHsc
#include <iostream>
using namespace std;

int main()
{
    int m = 0;
    int n = 0;
    [&, n] (int a) mutable { m = ++n + a; }(4);
    cout << m << endl << n << endl;
}
```

输出：

```
5
0
```

由于变量 **n** 是通过值捕获的，因此在调用 lambda 表达式后，变量的值仍保持 **0** 不变。**mutable** 规范允许在 lambda 中修改 **n**。

尽管 lambda 表达式只能捕获具有自动存储持续时间的变量，但你可以在 lambda 表达式的主体中使用具有静态存储持续时间的变量。以下示例使用 **generate** 函数和 lambda 表达式为 **vector** 对象中的每个元素赋值。lambda 表达式将修改静态变量以生成下一个元素的值。

**C++**

```
void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this is not thread-safe and is shown for illustration only
}
```

有关详细信息，请参阅[生成](#)。

下面的代码示例使用上一示例中的函数，并添加了使用 STL 算法 **generate\_n** 的 lambda 表达式的示例。该 lambda 表达式将 **vector** 对象的元素指派给前两个元素之和。使用了 **mutable** 关键字，以使 lambda 表达式的主体可以修改 lambda 表达式通过值捕获的外部变量 **x** 和 **y** 的副本。由于 lambda 表达式通过值捕获原始变量 **x** 和 **y**，因此它们的值在 lambda 执行后仍为 1。

C++

```
// compile with: /W4 /EHsc
#include <algorithm>
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this is not thread-safe and is shown for illustration only
}

int main()
{
    // The number of elements in the vector.
    const int elementCount = 9;

    // Create a vector object with each element set to 1.
    vector<int> v(elementCount, 1);

    // These variables hold the previous two elements of the vector.
    int x = 1;
```

```

int y = 1;

// Sets each element in the vector to the sum of the
// previous two elements.
generate_n(v.begin() + 2,
    elementCount - 2,
    [=]() mutable throw() -> int { // lambda is the 3rd parameter
        // Generate current value.
        int n = x + y;
        // Update previous two values.
        x = y;
        y = n;
        return n;
    });
print("vector v after call to generate_n() with lambda: ", v);

// Print the local variables x and y.
// The values of x and y hold their initial values because
// they are captured by value.
cout << "x: " << x << " y: " << y << endl;

// Fill the vector with a sequence of numbers
fillVector(v);
print("vector v after 1st call to fillVector(): ", v);
// Fill the vector with the next sequence of numbers
fillVector(v);
print("vector v after 2nd call to fillVector(): ", v);
}

```

输出：

```

vector v after call to generate_n() with lambda: 1 1 2 3 5 8 13 21 34
x: 1 y: 1
vector v after 1st call to fillVector(): 1 2 3 4 5 6 7 8 9
vector v after 2nd call to fillVector(): 10 11 12 13 14 15 16 17 18

```

有关详细信息，请参阅 [generate\\_n](#)。

## Microsoft 专用

以下公共语言运行时 (CLR) 托管实体中不支持 **lambda**：**ref class**、**ref struct**、**value class** 或 **value struct**。

若使用 Microsoft 专用的修饰符（例如 [\\_\\_declspec](#)），则你可以紧接在 parameter-declaration-clause 后将其插入到 **lambda** 表达式，例如：



```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t; };
```

若要确定 lambda 是否支持某个修饰符，请参阅文档的 [Microsoft 专用的修饰符](#) 部分中有关此内容的文章。

Visual Studio 支持 C++11 标准 lambda 表达式语法和功能，以下功能除外：

- 像所有其他类一样，lambda 不会获得自动生成的移动构造函数和移动赋值运算符。有关右值引用行为支持的详细信息，请参阅[支持 C++11/14/17 功能（现代 C++）](#) 中的“右值引用”部分。
- 此版本不支持可选的 attribute-specifier-seq。

除了 C++11 标准 lambda 功能之外，Visual Studio 还包括以下功能：

- 无状态 lambda，可完全转换为使用任意调用约定的函数指针。
- 只要所有返回语句具有相同的类型，就会自动推导比 { **return expression;** } 更复杂的 lambda 主体的返回类型。（此功能是拟建的 C++14 标准的一部分。）

## 请参见

### 参考

[函数调用 \(C++\)](#)

[for\\_each](#)

### 概念

[函数对象](#)

### 其他资源

[C++ 语言参考](#)