LAMBDA FUNCTIONS

C++11 provides the ability to create anonymous functions, called lambda functions. It allows a function to be defined at the point where it's needed in another expression. It is a function that we can write inline in our code in order to pass in to another function.

Previously, **for\_each()** needs a named function as shown in the example below:

/\* lambda.cpp \*/

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

// assign a value to each element of a vector

void assign(int& v)

{

static int n = 1; v = n++;

}

// print out each element

void print(int v)

{

cout << v << " ";

}

int main()

{

vector<int> vec(10);

// output initial value of each element

for\_each(vec.begin(), vec.end(), print);

cout << endl;

// assign a value to each element of a vector

for\_each(vec.begin(), vec.end(), assign);

// output updated value of each element

for\_each(vec.begin(), vec.end(), print);

return 0;

}

Output:

$ g++ -std=c++11 -o lambda lambda.cpp

$ ./lambda

0 0 0 0 0 0 0 0 0 0

1 2 3 4 5 6 7 8 9 10

Now, we can use **lambda functions [](){}** for the **print()** and **assign()** functions:

/\* lambda2.cpp \*/

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

int main()

{

vector<int> vec(10);

// output initial value of each element

// for\_each(vec.begin(), vec.end(), print); ==>

for\_each(vec.begin(), vec.end(), [](int v) {cout << v << " ";});

cout << endl;

// assign a value to each element of a vector

// for\_each(vec.begin(), vec.end(), assign); ==>

for\_each(vec.begin(), vec.end(), [](int& v) {static int n = 1; v = n++;});

// output updated value of each element

// for\_each(vec.begin(), vec.end(), print); ==>

for\_each(vec.begin(), vec.end(), [](int v) {cout << v << " ";});

return 0;

}

Output is the same as in the previous example:

$ g++ -std=c++11 -o lambda2 lambda2.cpp

$ ./lambda2

0 0 0 0 0 0 0 0 0 0

1 2 3 4 5 6 7 8 9 10

A lambda expression defines a self-contained function that takes no parameters and relies only on global variables and functions. It doesn't even have to return a value. Such a lambda expression is a series of statements enclosed in braces, prefixed with **[]**, called lambda **introducer** or **capture** specification which tells the compiler we're creating a lambda function, **[](){}**.

Let's look at lambda functions more useful:

#include <iostream>

#include <vector>

#include <algorithm>

void foo() { std::cout << "foo()\n"; }

void bar() { std::cout << "bar()\n"; }

int main ()

{

// 1st lambda function

auto f = []() {

foo();

bar();

};

// the lambda function does something here

f();

// 2nd lambda function

std::vector<int> v(5, 99);

std::for\_each(v.begin(), v.end(), [](int i){std::cout << i << "\n";});

return 0;

}

The output from the run:

foo()

bar()

99

99

99

99

99

The first lambda expression is rather unusual because it has no parameters inside the parentheses. If we need to take parameters, we can do this by following the lambda introducer with a parameter list just like for a normal function as shown in the second example which writes all the elements of the vector.

more examples

Here are more sample codes:

int main()

{

// (1)

std::cout << [](int a, int b){return a\*b; }(4, 5) << std::endl; // 20

// (2)

auto f = [](int a, int b) { return a\*b; };

std::cout << f(4, 5) << std::endl; // 20

}

The (1) and (2) are equivalent, and produced the results, 20.

Return from a lambda function

The compiler can deduce the return value type from a lambda function as shown in the case #1 of the example below. However, still we can explicitly specify its return type as in the case #2:

/\* lam.cpp \*/

#include <iostream>

using namespace std;

int main()

{

/\* case #1 - compiler deduces return type \*/

cout << [](int n) {return n\*n;} (5);

cout << endl;

/\* case #2 - explicit return type \*/

cout << [](int n)->int {return n\*n;} (5);

return 0;

}

Output:

$ g++ -std=c++11 -o lam lam.cpp

$ ./lam

25

25