C++ Lambda Expression

A lambda is a programming technique that's related to anonymous functions. A lambda implicitly defines a function object class and constructs a function object of that class type

A *function object*, or *functor*, is any type that implements operator(). This operator is referred to as the *call operator* or sometimes the *application operator*.

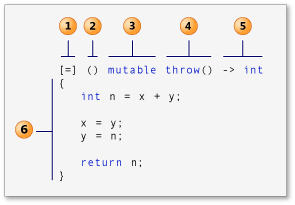
see remove\_if below

template<class ForwardIterator, class Predicate>  
 ForwardIterator remove\_if(  
 ForwardIterator \_First,  
 ForwardIterator \_Last,  
 Predicate \_Pred  
 );

in C#

public delegate TResult Func<in T, out TResult>(  
 T arg

The following illustration maps the grammar to the example.



The callouts in the illustration are as follows:

1. *lambda-introducer* (referred to as *capture clause* later in this topic)
2. *lambda-parameter-declaration-list* (referred to as *parameter list* later in this topic)
3. *mutable-specification* (referred to as *mutable specification* later in this topic)
4. *exception-specification* (referred to as *exception specification* later in this topic)
5. *lambda-return-type-clause* (referred to as *return type* later in this topic)
6. *compound-statement* (referred to as *lambda body* later in this topic)

The following sections describe the grammar in more detail.

**Capture Clause**

A lambda expression can access any variable that has automatic storage duration and that can be accessed in the enclosing scope. The capture clause specifies whether the body of the lambda expression accesses variables in the enclosing scope by value or by reference: variables that have the ampersand (&) prefix are accessed by reference and variables that do not have the & prefix are accessed by value. The empty capture clause, [], indicates that the body of the lambda expression accesses no variables in the enclosing scope.

You can specify the default capture mode (*capture-default* in the syntax) by specifying & or = as the first element of the capture clause. Below are equal:

**[&total, factor]  
[&, factor]  
[=, &total]**

**Parameter List**

The parameter list for a lambda expression resembles the parameter list for a function, with the following exceptions:

* The parameter list cannot have default arguments.
* The parameter list cannot have a variable-length argument list.
* The parameter list cannot have unnamed parameters.

int main()  
{  
 int x = 4;  
 int y = 5;  
 int z = [=] { return x + y; }(); //() means execute here  
}

**Exception Specification**

You can use the throw() exception specification to indicate that the lambda expression does not throw any exceptions. Similar to mutable which means can modify the external value copies.

**Return Type**

The return type part of a lambda expression resembles the return type part of an ordinary method or function. However, the return type follows the parameter list and you must include -> before the return type.

You can omit the return type part of a lambda expression if the lambda body contains a single return statement or the lambda expression does not return a value.

Lambda Body

A lambda expression can access the following types of variables:

* Parameters
* Locally-declared variables
* Class data members (when declared inside a class)
* Any variable that has static storage duration (for example, global variables)

# Declaring Lambda

// declaring\_lambda\_expressions1.cpp  
#include <functional>  
  
int main()  
{  
 // Assign the lambda expression that adds two numbers to an auto variable.  
 auto f1 = [] (int x, int y) { return x + y; };   
  
 // Assign the same lambda expression to a function object.  
 function<int (int, int)> f2 = [] (int x, int y) { return x + y; };  
}

The Visual C++ compiler binds a lambda expression to its captured variables when the expression is declared instead of when the expression is called.

# Calling Lambda

// Use the find\_if function and a lambda expression to find the   
 // first even number in the list.  
 const list<int>::const\_iterator result =  
 find\_if(numbers.begin(), numbers.end(),   
 [](int n) { return (n % 2) == 0; });

# Nest Lambda

// The following lambda expression contains a nested lambda  
 // expression.  
 int m = [](int x)   
 { return [](int y) { return y \* 2; }(x) + 3; }(5);

# Higher-Order Lambda Functions

A higher-order function is a lambda expression that takes another lambda expression as its argument or that returns a lambda expression.

// The following code declares a lambda expression that returns   
 // another lambda expression that adds two numbers.   
 // The returned lambda expression captures parameter x by value.  
 auto g = [](int x) -> function<int (int)>   
 { return [=](int y) { return x + y; }; };  
  
 // The following code declares a lambda expression that takes another  
 // lambda expression as its argument.  
 // The lambda expression applies the argument z to the function f  
 // and adds 1.  
 auto h = [](const function<int (int)>& f, int z)   
 { return f(z) + 1; };  
  
 // Call the lambda expression that is bound to h.   
 auto a = h(g(7), 8);

//euqals to g(x=7)(y=8) + 1;  
  
 // Print the result.  
 cout << a << endl;

# Use Lambda Expression in a Method

You can use the this pointer explicitly in a method, as shown in the following example:

void ApplyScale(const vector<int>& v) const  
{  
 for\_each(v.begin(), v.end(),   
 [this](int n) { cout << n \* this->\_scale << endl; });  
}

You can also capture the this pointer implicitly, as shown in the following example:

void ApplyScale(const vector<int>& v) const  
{  
 for\_each(v.begin(), v.end(),   
 [=](int n) { cout << n \* \_scale << endl; });  
}

# Use Lambda with Template

// Negates each element in the vector object.  
template <typename T>   
void negate\_all(vector<T>& v)  
{  
 for\_each(v.begin(), v.end(), [] (T& n) { n = -n; } );  
}

# Handle Exception

The body of a lambda expression follows the rules for both structured exception handling (SEH) and C++ exception handling.

try  
 {  
 for\_each(indices.begin(), indices.end(),   
 [&] (int index) { elements.at(index) = index; });  
 }  
 catch (const out\_of\_range& e)   
 {  
 cerr << "Caught '" << e.what() << "'." << endl;  
 };