

# Lambda Expressions in C++

11/18/2018 • 12 minutes to read • Contributors      all

## In this article

[Related Topics](#)

[Parts of a Lambda Expression](#)

[constexpr lambda expressions](#)

[Microsoft-Specific](#)

[See also](#)

In C++11 and later, a lambda expression—often called a *lambda*—is a convenient way of defining an anonymous function object (a *closure*) right at the location where it is invoked or passed as an argument to a function. Typically lambdas are used to encapsulate a few lines of code that are passed to algorithms or asynchronous methods. This article defines what lambdas are, compares them to other programming techniques, describes their advantages, and provides a basic example.

## Related Topics

- [Lambda expressions vs. function objects](#)
- [Working with lambda expressions](#)
- [constexpr lambda expressions](#)

## Parts of a Lambda Expression

The ISO C++ Standard shows a simple lambda that is passed as the third argument to the `std::sort()` function:

C++

 Copy

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

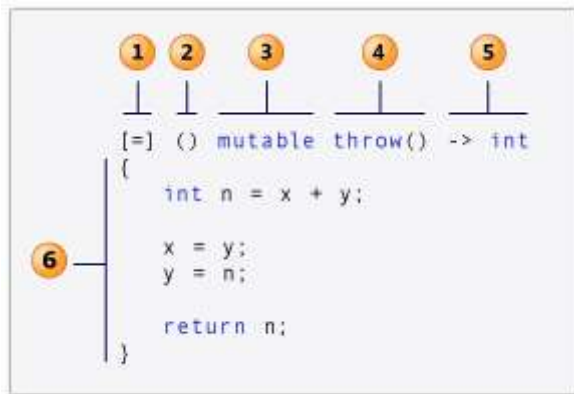
void absort(float* x, unsigned n) {
    std::sort(x, x + n,
        // Lambda expression begins
        [](float a, float b) {
            return (std::abs(a) < std::abs(b));
        });
}
```

```

    } // end of lambda expression
  );
}

```

This illustration shows the parts of a lambda:



1. *capture clause* (Also known as the *lambda-introducer* in the C++ specification.)
2. *parameter list* Optional. (Also known as the *lambda declarator*)
3. *mutable specification* Optional.
4. *exception-specification* Optional.
5. *trailing-return-type* Optional.
6. *lambda body*.

## Capture Clause

A lambda can introduce new variables in its body (in C++14), and it can also access, or *capture*, variables from the surrounding scope. A lambda begins with the capture clause (*lambda-introducer* in the Standard syntax), which specifies which variables are captured, and whether the capture is by value or by reference. Variables that have the ampersand ( `&` ) prefix are accessed by reference and variables that do not have it are accessed by value.

An empty capture clause, `[ ]`, indicates that the body of the lambda expression accesses no variables in the enclosing scope.

You can use the default capture mode (*capture-default* in the Standard syntax) to indicate how to capture any outside variables that are referenced in the lambda: `[&]` means all variables that you refer to are captured by reference, and `[=]` means they are captured by

value. You can use a default capture mode, and then specify the opposite mode explicitly for specific variables. For example, if a lambda body accesses the external variable `total` by reference and the external variable `factor` by value, then the following capture clauses are equivalent:

C++

 Copy

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

Only variables that are mentioned in the lambda are captured when a capture-default is used.

If a capture clause includes a capture-default `&`, then no `identifier` in a `capture` of that capture clause can have the form `& identifier`. Likewise, if the capture clause includes a capture-default `=`, then no `capture` of that capture clause can have the form `= identifier`. An identifier or **this** cannot appear more than once in a capture clause. The following code snippet illustrates some examples.

C++

 Copy

```
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
    [=, *this]{};        // OK: captures this by value. See below.
    [i, i]{};           // ERROR: i repeated
}
```

A capture followed by an ellipsis is a pack expansion, as shown in this [variadic template](#) example:

C++

 Copy

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

```
x();  
}
```

To use lambda expressions in the body of a class method, pass the **this** pointer to the capture clause to provide access to the methods and data members of the enclosing class.

**Visual Studio 2017 version 15.3 and later** (available with [/std:c++17](#)): The **this** pointer may be captured by value by specifying `*this` in the capture clause. Capture by value means that the entire *closure*, which is the anonymous function object that encapsulates the lambda expression, is copied to every call site where the lambda is invoked. Capture by value is useful when the lambda will execute in parallel or asynchronous operations, especially on certain hardware architectures such as NUMA.

For an example that shows how to use lambda expressions with class methods, see "Example: Using a Lambda Expression in a Method" in [Examples of Lambda Expressions](#).

When you use the capture clause, we recommend that you keep these points in mind, particularly when you use lambdas with multithreading:

- Reference captures can be used to modify variables outside, but value captures cannot. (**mutable** allows copies to be modified, but not originals.)
- Reference captures reflect updates to variables outside, but value captures do not.
- Reference captures introduce a lifetime dependency, but value captures have no lifetime dependencies. This is especially important when the lambda runs asynchronously. If you capture a local by reference in an async lambda, that local will very possibly be gone by the time the lambda runs, resulting in an access violation at run time.

## Generalized capture (C++ 14)

In C++14, you can introduce and initialize new variables in the capture clause, without the need to have those variables exist in the lambda function's enclosing scope. The initialization can be expressed as any arbitrary expression; the type of the new variable is deduced from the type produced by the expression. One benefit of this feature is that in C++14 you can capture move-only variables (such as `std::unique_ptr`) from the surrounding scope and use them in a lambda.

C++

 Copy

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

## Parameter List

In addition to capturing variables, a lambda can accept input parameters. A parameter list (*lambda declarator* in the Standard syntax) is optional and in most aspects resembles the parameter list for a function.

C++

 Copy

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

In **C++ 14**, if the parameter type is generic, you can use the `auto` keyword as the type specifier. This tells the compiler to create the function call operator as a template. Each instance of `auto` in a parameter list is equivalent to a distinct type parameter.

C++

 Copy

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

A lambda expression can take another lambda expression as its argument. For more information, see "Higher-Order Lambda Expressions" in the topic [Examples of Lambda Expressions](#).

Because a parameter list is optional, you can omit the empty parentheses if you do not pass arguments to the lambda expression and its lambda-declarator does not contain *exception-specification*, *trailing-return-type*, or **mutable**.

## Mutable Specification

Typically, a lambda's function call operator is const-by-value, but use of the **mutable** keyword cancels this out. It does not produce mutable data members. The mutable specification enables the body of a lambda expression to modify variables that are captured by value. Some of the examples later in this article show how to use **mutable**.

## Exception Specification

You can use the `noexcept` exception specification to indicate that the lambda expression does not throw any exceptions. As with ordinary functions, the Visual C++ compiler generates warning [C4297](#) if a lambda expression declares the `noexcept` exception specification and the lambda body throws an exception, as shown here:


C++	 Copy
<pre>// throw_lambda_expression.cpp // compile with: /W4 /EHsc int main() // C4297 expected {     []() noexcept { throw 5; }(); }</pre>	

For more information, see [Exception Specifications \(throw\)](#).

## Return Type

The return type of a lambda expression is automatically deduced. You don't have to use the [auto](#) keyword unless you specify a *trailing-return-type*. The *trailing-return-type* resembles the return-type part of an ordinary method or function. However, the return type must follow the parameter list, and you must include the trailing-return-type keyword `->` before the return type.

You can omit the return-type part of a lambda expression if the lambda body contains just one return statement or the expression does not return a value. If the lambda body contains one return statement, the compiler deduces the return type from the type of the return expression. Otherwise, the compiler deduces the return type to be **void**. Consider the following example code snippets that illustrate this principle.

C++	 Copy
<pre>auto x1 = [](int i){ return i; }; // OK: return type is int auto x2 = []{ return{ 1, 2 }; }; // ERROR: return type is void, deducing</pre>	

```
valid // return type from braced-init-list is not
```

A lambda expression can produce another lambda expression as its return value. For more information, see "Higher-Order Lambda Expressions" in [Examples of Lambda Expressions](#).

## Lambda Body

The lambda body (*compound-statement* in the Standard syntax) of a lambda expression can contain anything that the body of an ordinary method or function can contain. The body of both an ordinary function and a lambda expression can access these kinds of variables:

- Captured variables from the enclosing scope, as described previously.
- Parameters
- Locally-declared variables
- Class data members, when declared inside a class and **this** is captured
- Any variable that has static storage duration—for example, global variables

The following example contains a lambda expression that explicitly captures the variable `n` by value and implicitly captures the variable `m` by reference:

C++

 Copy

```
// 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;
}
```

Output

 Copy

```
5
0
```

Because the variable `n` is captured by value, its value remains `0` after the call to the lambda expression. The **mutable** specification allows `n` to be modified within the lambda.

Although a lambda expression can only capture variables that have automatic storage duration, you can use variables that have static storage duration in the body of a lambda expression. The following example uses the `generate` function and a lambda expression to assign a value to each element in a `vector` object. The lambda expression modifies the static variable to generate the value of the next element.

C++

 Copy

```
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
}
```

For more information, see [generate](#).

The following code example uses the function from the previous example, and adds an example of a lambda expression that uses the C++ Standard Library algorithm `generate_n`. This lambda expression assigns an element of a `vector` object to the sum of the previous two elements. The **mutable** keyword is used so that the body of the lambda expression can modify its copies of the external variables `x` and `y`, which the lambda expression captures by value. Because the lambda expression captures the original variables `x` and `y` by value, their values remain `1` after the lambda executes.

C++

 Copy

```
// 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);
}
```

Output

 Copy

```
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
```

For more information, see [generate\\_n](#).

## constexpr lambda expressions

Visual Studio 2017 version 15.3 and later (available with [/std:c++17](#)): A lambda expression may be declared as `constexpr` or used in a constant expression when the initialization of each data member that it captures or introduces is allowed within a constant expression.

C++

 Copy

```
int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
}
```

A lambda is implicitly `constexpr` if its result satisfies the requirements of a `constexpr` function:

C++

 Copy

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

If a lambda is implicitly or explicitly `constexpr`, conversion to a function pointer produces a `constexpr` function:

C++

 Copy

```
auto Increment = [](int n)
{
    return n + 1;
};


constexpr int(*inc)(int) = Increment;
```

## Microsoft-Specific

Lambdas are not supported in the following common language runtime (CLR) managed entities: **ref class**, **ref struct**, **value class**, or **value struct**.

If you are using a Microsoft-specific modifier such as [\\_\\_declspec](#), you can insert it into a lambda expression immediately after the `parameter-declaration-clause`—for example:

C++

 Copy

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

To determine whether a modifier is supported by lambdas, see the article about it in the [Microsoft-Specific Modifiers](#) section of the documentation.

In addition to C++11 Standard lambda functionality, Visual Studio supports stateless lambdas, which are omni-convertible to function pointers that use arbitrary calling conventions.

## See also

[C++ Language Reference](#)

[Function Objects in the C++ Standard Library.](#)

[Function Call](#)

[for\\_each](#)