Lambda expressions (since C++11)

Constructs a closure : an unnamed function object capable of capturing variables in scope.

Syntax

Lambda expressions without an explicit template parameter list (possibly non-generic)

[*captures* **]** *front-attr*(optional) **(** *params* **)** *specs*(optional) *exception*(optional) *back-attr*(optional) *trailing-type*(optional) *requires*(optional) **{** *body* **}**	(1)	
[*captures* **]** **{** *body* **}**	(2)	(until C++23)
[*captures* **]** *front-attr*(optional) *trailing-type*(optional) **{** *body* **}**	(2)	(since C++23)
[*captures* **]** *front-attr*(optional) *exception* *back-attr*(optional) *trailing-type*(optional) **{** *body* **}**	(3)	(since C++23)
[*captures* **]** *front-attr*(optional) *specs* *exception*(optional) *back-attr*(optional) *trailing-type*(optional) **{** *body* **}**	(4)	(since C++23)

Lambda expressions with an explicit template parameter list (always generic) (since C++20)

[*captures* **]** **<** *tparams* **>** *t-requires*(optional) *front-attr*(optional) **(** *params* **)** *specs*(optional) *exception*(optional) *back-attr*(optional) *trailing-type*(optional) *requires*(optional) **{** *body* **}**	(1)	
[*captures* **]** **<** *tparams* **>** *t-requires*(optional) **{** *body* **}**	(2)	(until C++23)
[*captures* **]** **<** *tparams* **>** *t-requires*(optional) *front-attr*(optional) *trailing-type*(optional) **{** *body* **}**	(2)	(since C++23)
[*captures* **]** **<** *tparams* **>** *t-requires*(optional) *front-attr*(optional) *exception* *back-attr*(optional) *trailing-type*(optional) **{** *body* **}**	(3)	(since C++23)
[*captures* **]** **<** *tparams* **>** *t-requires*(optional) *front-attr*(optional) *specs* *exception*(optional) *back-attr*(optional) *trailing-type*(optional) **{** *body* **}**	(4)	(since C++23)

1) The lambda expression with a parameter list.

2-4) The lambda expression without a parameter list.

2) The simplest syntax. *back-attr* cannot be applied.

3,4) *back-attr* can only be applied if any of *specs* and *exception* is present.

Explanation

captures - A comma-separated list of zero or more captures, optionally beginning with a *capture-default*.

See below for the detailed description of captures.

A lambda expression can use a variable without capturing it if the variable

- is a non-local variable or has static or thread local storage duration (in which case the variable cannot be captured), or
- is a reference that has been initialized with a constant expression.

A lambda expression can read the value of a variable without capturing it if the variable

- has const non-volatile integral or enumeration type and has been initialized with a constant expression, or
- is constexpr and has no mutable members.

tparams - A non-empty comma-separated list of template parameters, used to provide names to the template parameters of a generic lambda (see `ClosureType::operator()` below).

t-requires - Adds constraints to *tparams*.

front-attr - (since C++23) An attribute specifier sequence applies to `operator()` of the closure type (and thus the `[[noreturn]]` attribute can be used).

params - The parameter list of `operator()` of the closure type.

specs - A list of the following specifiers, each specifier is allowed at most once in each sequence.

Specifier	Effect
`mutable`	Allows *body* to modify the objects captured by copy, and to call their non-const member functions.
`constexpr` (since C++17)	Explicitly specifies that `operator()` is a constexpr function. - If `operator()` satisfy all constexpr function requirements, `operator()` will be constexpr even if `constexpr` is not present.
`consteval` (since C++20)	Specifies that `operator()` is an immediate function. - `consteval` and `constexpr` cannot be specified at the same time.
`static` (since C++23)	Specifies that `operator()` is a static member function. - `static` and `mutable` cannot be specified at the same time. - Cannot be used if *captures* is not empty, or an explicit object parameter is present.

exception - Provides the dynamic exception specification or the noexcept specifier for `operator()` of the closure type.

back-attr - An attribute specifier sequence applies to the type of `operator()` of the closure type (and thus the `[[noreturn]]` attribute cannot be used).

trailing-type - **->** *ret*, where *ret* specifies the return type.

requires - (since C++20) Adds constraints to `operator()` of the closure type.

body - The function body.

A variable `__func__` is implicitly defined at the beginning of *body*, with semantics as described here.

Closure type

The lambda expression is a prvalue expression of unique unnamed non-union non-aggregate class type, known as *closure type*, which is declared (for the purposes of ADL) in the smallest block scope, class scope, or namespace scope that contains the lambda expression.

The closure type has the following members, they cannot be named in a friend declaration:

ClosureType::operator()(*params*)

```
ret operator()(params) { body }
```

(static and const may be present, see below)

Executes the body of the lambda expression, when invoked. When accessing a variable, accesses its captured copy (for the entities captured by copy), or the original object (for the entities captured by reference).

The parameter list of `operator()` is *params* if it is provided, otherwise the parameter list is empty.

The return type of `operator()` is the type specified in *trailing-type*.

If *trailing-type* is not provided, the return type of `operator()` is automatically deduced.[1]

Unless the keyword `mutable` was used in the lambda specifiers, the cv-qualifier of `operator()` is `const` and the objects that were captured by copy are non-modifiable from inside this `operator()`. Explicit `const` qualifier is not allowed. `operator()` is never virtual and cannot have the `volatile` qualifier.

The exception specification *exception* on the lambda expression applies to `operator()`.

For the purpose of name lookup, determining the type and value of the `this` pointer and for accessing non-static class members, the body of the closure type's `operator()` is considered in the context of the lambda expression.

```
struct X
{
    int x, y;
    int operator()(int);
    void f()
    {
        // the context of the following lambda is the member function X::f
        [=]() -> int
        {
            return operator()(this->x + y); // X::operator()(this->x + (*this).y)
                                            // this has type X*
        };
    }
};
```

Dangling references

If a non-reference entity is captured by reference, implicitly or explicitly, and `operator()` of the closure object is invoked after the entity's lifetime has ended, undefined behavior occurs. The C++ closures do not extend the lifetimes of objects captured by reference.

Same applies to the lifetime of the current `*this` object captured via **`this`**.

1. ↑ Although function return type deduction is introduced in C++14, its rule is available for lambda return type deduction in C++11.

ClosureType::operator *ret*(*)(*params*)()

capture-less non-generic lambda

```
using F = ret(*)(params);
operator F() const noexcept;
```

capture-less generic lambda

This user-defined conversion function is only defined if the capture list of the lambda expression is empty. It is a public, non-virtual, non-explicit, const noexcept member function of the closure object.

The value returned by the conversion function is a pointer to a function with C++ language linkage that, when invoked, has the same effect as invoking the closure type's function call operator on a default-constructed instance of the closure type.

ClosureType::ClosureType()

```
ClosureType(const ClosureType&) = default;
```

```
ClosureType(ClosureType&&) = default;
```

Closure types are not *DefaultConstructible*. Closure types have no default constructor.

The copy constructor and the move constructor are declared as defaulted and may be implicitly-defined according to the usual rules for copy constructors and move constructors.

ClosureType::operator=(const ClosureType&)

```
ClosureType& operator=(const ClosureType&) = delete;
```

The copy assignment operator is defined as deleted (and the move assignment operator is not declared). Closure types are not *CopyAssignable*.

ClosureType::~ClosureType()

```
~ClosureType() = default;
```

The destructor is implicitly-declared.

ClosureType::*Captures*

```
T1 a;
T2 b;
...
```

If the lambda expression captures anything by copy (either implicitly with capture clause **`[=]`** or explicitly with a capture that does not include the character &, e.g. **`[a, b, c]`**), the closure type includes unnamed non-static data members, declared in unspecified order, that hold copies of all entities that were so captured.

Those data members that correspond to captures without initializers are direct-initialized when the lambda expression is evaluated. Those that correspond to captures with initializers are initialized as the initializer requires (could be copy- or direct-initialization). If an array is captured, array elements are direct-initialized in increasing index order. The order in which the data members are initialized is the order in which they are declared (which is unspecified).

The type of each data member is the type of the corresponding captured entity, except if the entity has reference type (in that case, references to functions are captured as lvalue references to the referenced functions, and references to objects are captured as copies of the referenced objects).

For the entities that are captured by reference (with the capture-default **[&]** or when using the character &, e.g. **[&a, &b, &c]**), it is unspecified if additional data members are declared in the closure type.

Lambda expressions are not allowed in unevaluated expressions, template arguments, alias declarations, typedef declarations, and anywhere in a function (or function template) declaration except the function body and the function's default arguments.

Lambda capture

The *captures* is a comma-separated list of zero or more *captures*, optionally beginning with the *capture-default*. The capture list defines the outside variables that are accessible from within the lambda function body. The only capture defaults are

- **&** (implicitly capture the used variables with automatic storage duration by reference) and
- **=** (implicitly capture the used variables with automatic storage duration by copy).

The current object (`*this`) can be implicitly captured if either capture default is present. If implicitly captured, it is always captured by reference, even if the capture default is **=**.

The syntax of an individual capture in *captures* is

identifier	(1)	
identifier **...**	(2)	
identifier initializer	(3)	(since C++14)
& *identifier*	(4)	
& *identifier* **...**	(5)	
& *identifier initializer*	(6)	(since C++14)
this	(7)	
***** **this**	(8)	(since C++17)
... *identifier initializer*	(9)	(since C++20)
& ... *identifier initializer*	(10)	(since C++20)

1) simple by-copy capture
2) simple by-copy capture that is a pack expansion
3) by-copy capture with an initializer
4) simple by-reference capture
5) simple by-reference capture that is a pack expansion
6) by-reference capture with an initializer
7) simple by-reference capture of the current object
8) simple by-copy capture of the current object
9) by-copy capture with an initializer that is a pack expansion
10) by-reference capture with an initializer that is a pack expansion

If the capture-default is **&**, subsequent simple captures must not begin with **&**.

```
struct S2 { void f(int i); };
void S2::f(int i)
{
    [&] {};          // OK: by-reference capture default
    [&, i] {};       // OK: by-reference capture, except i is captured by copy
    [&, &i] {};      // Error: by-reference capture when by-reference is the default
    [&, this] {};    // OK, equivalent to [&]
    [&, this, i] {}; // OK, equivalent to [&, i]
}
```

If the capture-default is **=**, subsequent simple captures must begin with **&** .

```
struct S2 { void f(int i); };
void S2::f(int i)
{
    [=] {};        // OK: by-copy capture default
    [=, &i] {};    // OK: by-copy capture, except i is captured by reference
    [=, *this] {}; // until C++17: Error: invalid syntax
                   // since C++17: OK: captures the enclosing S2 by copy
    [=, this] {};  // until C++20: Error: this when = is the default
```

```
                   // since C++20: OK, same as [=]
}
```

Any capture may appear only once, and its name must be different from any parameter name:

```
struct S2 { void f(int i); };
void S2::f(int i)
{
    [i, i] {};        // Error: i repeated
    [this, *this] {}; // Error: "this" repeated (C++17)

    [i] (int i) {};   // Error: parameter and capture have the same name
}
```

Only lambda expressions defined at block scope or in a default member initializer may have a capture-default or captures without initializers. For such lambda expression, the *reaching scope* is defined as the set of enclosing scopes up to and including the innermost enclosing function (and its parameters). This includes nested block scopes and the scopes of enclosing lambdas if this lambda is nested.

The *identifier* in any capture without an initializer (other than the `this`-capture) is looked up using usual unqualified name lookup in the *reaching scope* of the lambda. The result of the lookup must be a variable with automatic storage duration declared in the reaching scope. The entity is *explicitly captured*.

If a capture list has a capture-default and does not explicitly capture the enclosing object (as `this` or `*this`), or an automatic variable that is odr-usable in the lambda body, it captures the entity *implicitly* if the entity is named in a potentially-evaluated expression within an expression (including when the implicit `this->` is added before a use of non-static class member).

For the purpose of determining implicit captures, `typeid` is never considered to make its operands unevaluated.

```
void f(int, const int (&)[2] = {}) {}   // #1
void f(const int&, const int (&)[1]) {} // #2

struct NoncopyableLiteralType
{
    constexpr explicit NoncopyableLiteralType(int n) : n_(n) {}
    NoncopyableLiteralType(const NoncopyableLiteralType&) = delete;

    int n_;
};

void test()
{
    const int x = 17;

    auto l0 = []{ f(x); };           // OK: calls #1, does not capture x
    auto g0 = [](auto a) { f(x); };  // same as above

    auto l1 = [=]{ f(x); };          // OK: captures x (since P0588R1) and calls #1
                                     // the capture can be optimized away
    auto g1 = [=](auto a) { f(x); }; // same as above

    auto ltid = [=]{ typeid(x); };   // OK: captures x (since P0588R1)
                                     // even though x is unevaluated
                                     // the capture can be optimized away

    auto g2 = [=](auto a)
    {
        int selector[sizeof(a) == 1 ? 1 : 2] = {};
        f(x, selector); // OK: is a dependent expression, so captures x
    };

    auto g3 = [=](auto a)
    {
        typeid(a + x);  // captures x regardless of
                        // whether a + x is an unevaluated operand
    };

    constexpr NoncopyableLiteralType w{42};
    auto l4 = []{ return w.n_; };      // OK: w is not odr-used, capture is unnecessary
    // auto l5 = [=]{ return w.n_; };  // error: w needs to be captured by copy
}
```

If the body of a lambda odr-uses an entity captured by copy, the member of the closure type is accessed. If it is not odr-using the entity, the access is to the original object:

```
void f(const int*);
void g()
{
    const int N = 10;
    [=]
    {
        int arr[N]; // not an odr-use: refers to g's const int N
        f(&N); // odr-use: causes N to be captured (by copy)
```

```
                // &N is the address of the closure object's member N, not g's N
    }();
}
```

If a lambda odr-uses a reference that is captured by reference, it is using the object referred-to by the original reference, not the captured reference itself:

Run this code

```
#include <iostream>

auto make_function(int& x)
{
    return [&] { std::cout << x << '\n'; };
}

int main()
{
    int i = 3;
    auto f = make_function(i); // the use of x in f binds directly to i
    i = 5;
    f(); // OK: prints 5
}
```

Within the body of a lambda with capture default **=**, the type of any capturable entity is as if it were captured (and thus const-qualification is often added if the lambda is not `mutable`), even though the entity is in an unevaluated operand and not captured (e.g. in `decltype`):

```
void f3()
{
    float x, &r = x;
    [=]
    { // x and r are not captured (appearance in a decltype operand is not an odr-use)
        decltype(x) y1;        // y1 has type float
        decltype((x)) y2 = y1; // y2 has type float const& because this lambda
                               // is not mutable and x is an lvalue
        decltype(r) r1 = y1;   // r1 has type float& (transformation not considered)
        decltype((r)) r2 = y2; // r2 has type float const&
    };
}
```

Any entity captured by a lambda (implicitly or explicitly) is odr-used by the lambda expression (therefore, implicit capture by a nested lambda triggers implicit capture in the enclosing lambda).

All implicitly-captured variables must be declared within the *reaching scope* of the lambda.

If a lambda captures the enclosing object (as `this` or `*this`), either the nearest enclosing function must be a non-static member function or the lambda must be in a default member initializer:

```
struct s2
{
    double ohseven = .007;
    auto f() // nearest enclosing function for the following two lambdas
    {
        return [this]      // capture the enclosing s2 by reference
        {
            return [*this] // capture the enclosing s2 by copy (C++17)
            {
                return ohseven; // OK
            }
        }();
    }

    auto g()
    {
        return [] // capture nothing
        {
            return [*this] {}; // error: *this not captured by outer lambda expression
        }();
    }
};
```

If a lambda expression ODR-uses `*this` or any variable with automatic storage duration, it must be captured by the lambda expression.

```
void f1(int i)
{
    int const N = 20;
    auto m1 = [=]
    {
        int const M = 30;
        auto m2 = [i]
        {
            int x[N][M]; // N and M are not odr-used
                         // (ok that they are not captured)
            x[0][0] = i; // i is explicitly captured by m2
                         // and implicitly captured by m1
        };
    };

    struct s1 // local class within f1()
    {
        int f;
        void work(int n) // non-static member function
        {
            int m = n * n;
            int j = 40;
            auto m3 = [this, m]
            {
                auto m4 = [&, j] // error: j is not captured by m3
                {
                    int x = n; // error: n is implicitly captured by m4
                               // but not captured by m3
                    x += m;    // OK: m is implicitly captured by m4
                               // and explicitly captured by m3
                    x += i;    // error: i is outside of the reaching scope
                               // (which ends at work())
                    x += f;    // OK: this is captured implicitly by m4
                               // and explicitly captured by m3
                };
            };
        }
    };
}
```

Class members cannot be captured explicitly by a capture without initializer (as mentioned above, only variables are permitted in the capture list):

```
class S
{
    int x = 0;

    void f()
    {
        int i = 0;
    //  auto l1 = [i, x] { use(i, x); };      // error: x is not a variable
        auto l2 = [i, x = x] { use(i, x); };  // OK, copy capture
        i = 1; x = 1; l2(); // calls use(0,0)
        auto l3 = [i, &x = x] { use(i, x); }; // OK, reference capture
        i = 2; x = 2; l3(); // calls use(1,2)
    }
};
```

When a lambda captures a member using implicit by-copy capture, it does not make a copy of that member variable: the use of a member variable `m` is treated as an expression `(*this).m`, and `*this` is always implicitly captured by reference:

```
class S
{
    int x = 0;

    void f()
    {
        int i = 0;

        auto l1 = [=] { use(i, x); }; // captures a copy of i and
                                      // a copy of the this pointer
        i = 1; x = 1; l1();           // calls use(0, 1), as if
                                      // i by copy and x by reference

        auto l2 = [i, this] { use(i, x); }; // same as above, made explicit
        i = 2; x = 2; l2();           // calls use(1, 2), as if
                                      // i by copy and x by reference

        auto l3 = [&] { use(i, x); }; // captures i by reference and
                                      // a copy of the this pointer
        i = 3; x = 2; l3();           // calls use(3, 2), as if
                                      // i and x are both by reference

        auto l4 = [i, *this] { use(i, x); }; // makes a copy of *this,
                                             // including a copy of x
        i = 4; x = 4; l4();           // calls use(3, 2), as if
                                      // i and x are both by copy
    }
};
```

If a lambda expression appears in a default argument, it cannot explicitly or implicitly capture anything:

```
void f2()
{
    int i = 1;

    void g1( int = [i] { return i; }() ); // error: captures something
    void g2( int = [i] { return 0; }() ); // error: captures something
    void g3( int = [=] { return i; }() ); // error: captures something

    void g4( int = [=] { return 0; }() );       // OK: capture-less
    void g5( int = [] { return sizeof i; }() ); // OK: capture-less

    // C++14
    void g6( int = [x = 1] { return x; }() ); // OK: 1 can appear
                                              //     in a default argument
    void g7( int = [x = i] { return x; }() ); // error: i cannot appear
                                              //        in a default argument
}
```

Members of anonymous unions members cannot be captured. Bit-fields can only be captured by copy.

If a nested lambda `m2` captures something that is also captured by the immediately enclosing lambda `m1`, then `m2`'s capture is transformed as follows:

- if the enclosing lambda `m1` captures by copy, `m2` is capturing the non-static member of `m1`'s closure type, not the original variable or `*this`; if `m1` is not mutable, the non-static data member is considered to be const-qualified.
- if the enclosing lambda `m1` captures by reference, `m2` is capturing the original variable or `*this`.

Run this code

```
#include <iostream>

int main()
{
    int a = 1, b = 1, c = 1;

    auto m1 = [a, &b, &c]() mutable
    {
        auto m2 = [a, b, &c]() mutable
        {
            std::cout << a << b << c << '\n';
            a = 4; b = 4; c = 4;
        };
        a = 3; b = 3; c = 3;
        m2();
    };

    a = 2; b = 2; c = 2;

    m1();                             // calls m2() and prints 123
    std::cout << a << b << c << '\n'; // prints 234
}
```

Notes

Feature-test macro	Value	Std	Feature
`__cpp_lambdas`	200907L	(C++11)	Lambda expressions
`__cpp_generic_lambdas`	201304L	(C++14)	Generic lambda expressions
	201707L	(C++20)	Explicit template parameter list for generic lambdas
`__cpp_init_captures`	201304L	(C++14)	Lambda init-capture
	201803L	(C++20)	Allow pack expansion in lambda init-capture
`__cpp_capture_star_this`	201603L	(C++17)	Lambda capture of `*this` by value as `[=, *this]`
`__cpp_constexpr`	201603L	(C++17)	constexpr lambda
`__cpp_static_call_operator`	202207L	(C++23)	static `operator()` for capture-less lambdas

The rule for implicit lambda capture is slightly changed by defect report P0588R1 (https://wg21.link/P0588R1) . As of 2023-10, some major implementations have not completely implemented the DR, and thus the old rule, which detects odr-using, is still used in some cases.

Old rule before P0588R1 **[Expand]**

Example

This example shows how to pass a lambda to a generic algorithm and how objects resulting from a lambda expression can be stored in `std::function` objects.

Run this code

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

int main()
{
    std::vector<int> c{1, 2, 3, 4, 5, 6, 7};
    int x = 5;
    c.erase(std::remove_if(c.begin(), c.end(), [x](int n) { return n < x; }), c.end());

    std::cout << "c: ";
    std::for_each(c.begin(), c.end(), [](int i) { std::cout << i << ' '; });
    std::cout << '\n';

    // the type of a closure cannot be named, but can be inferred with auto
    // since C++14, lambda could own default arguments
    auto func1 = [](int i = 6) { return i + 4; };
    std::cout << "func1: " << func1() << '\n';

    // like all callable objects, closures can be captured in std::function
    // (this may incur unnecessary overhead)
    std::function<int(int)> func2 = [](int i) { return i + 4; };
    std::cout << "func2: " << func2(6) << '\n';

    constexpr int fib_max {8};
    std::cout << "Emulate `recursive lambda` calls:\nFibonacci numbers: ";
    auto nth_fibonacci = [](int n)
    {
        std::function<int(int, int, int)> fib = [&](int n, int a, int b)
        {
            return n ? fib(n - 1, a + b, a) : b;
        };
        return fib(n, 0, 1);
    };

    for (int i{1}; i <= fib_max; ++i)
        std::cout << nth_fibonacci(i) << (i < fib_max ? ", " : "\n");

    std::cout << "Alternative approach to lambda recursion:\nFibonacci numbers: ";
    auto nth_fibonacci2 = [](auto self, int n, int a = 0, int b = 1) -> int
    {
        return n ? self(self, n - 1, a + b, a) : b;
    };

    for (int i{1}; i <= fib_max; ++i)
        std::cout << nth_fibonacci2(nth_fibonacci2, i) << (i < fib_max ? ", " : "\n");

#ifdef __cpp_explicit_this_parameter
    std::cout << "C++23 approach to lambda recursion:\n";
    auto nth_fibonacci3 = [](this auto self, int n, int a = 0, int b = 1) -> int
    {
         return n ? self(n - 1, a + b, a) : b;
    };

    for (int i{1}; i <= fib_max; ++i)
        std::cout << nth_fibonacci3(i) << (i < fib_max ? ", " : "\n");
#endif
}
```

Possible output:

```
c: 5 6 7
func1: 10
func2: 10
Emulate `recursive lambda` calls:
Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13
Alternative approach to lambda recursion:
Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13
```

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
CWG 974 (https://cplusplus.github.io/CWG/issues/974.html)	C++11	default argument was not allowed in the parameter list of a lambda expression	allowed
CWG 1048 (https://cplusplus.github.io/CWG/issues/1048.html) (N3638 (https://wg21.link/N3638))	C++11	the return type could only be deduced for lambda bodies containing only one `return` statement	improved the return type deduction
CWG 1249 (https://cplusplus.github.io/CWG/issues/1249.html)	C++11	it is not clear that whether the captured member of the enclosing non-mutable lambda is considered `const` or not	considered `const`
CWG 1557 (https://cplusplus.github.io/CWG/issues/1557.html)	C++11	the language linkage of the returned function type of the closure type's conversion function was not specified	it has C++ language linkage
CWG 1607 (https://cplusplus.github.io/CWG/issues/1607.html)	C++11	lambda expressions could appear in function and function template signatures	not allowed

CWG 1612 (https://cplusplus.github.io/CWG/issues/1612.html)	C++11	members of anonymous unions could be captured	not allowed
CWG 1722 (https://cplusplus.github.io/CWG/issues/1722.html)	C++11	the conversion function for capture-less lambdas had unspecified exception specification	conversion function is noexcept
CWG 1772 (https://cplusplus.github.io/CWG/issues/1772.html)	C++11	the semantic of `__func__` in lambda body was not clear	it refers to the closure class's operator()
CWG 1780 (https://cplusplus.github.io/CWG/issues/1780.html)	C++14	it was unclear whether the members of the closure types of generic lambdas can be explicitly instantiated or explicitly specialized	neither is allowed
CWG 1891 (https://cplusplus.github.io/CWG/issues/1891.html)	C++11	closure had a deleted default constructor and implicit copy/move constructors	no default and defaulted copy/move constructors
CWG 1937 (https://cplusplus.github.io/CWG/issues/1937.html)	C++11	as for the effect of invoking the result of the conversion function, it was unspecified on which object calling its `operator()` has the same effect	on a default-constructed instance of the closure type
CWG 1973 (https://cplusplus.github.io/CWG/issues/1973.html)	C++11	the parameter list of the closure type's `operator()` could refer to the parameter list given in *trailing-type*	can only refer to *params*
CWG 2011 (https://cplusplus.github.io/CWG/issues/2011.html)	C++11	for a reference captured by reference, it was unspecified which entity the identifier of the capture refers to	it refers to the originally referenced entity
CWG 2095 (https://cplusplus.github.io/CWG/issues/2095.html)	C++11	the behavior of capturing rvalue references to functions by copy was not clear	made clear
CWG 2211 (https://cplusplus.github.io/CWG/issues/2211.html)	C++11	the behavior was unspecified if a capture has the same name as a parameter	the program is ill-formed in this case
CWG 2358 (https://cplusplus.github.io/CWG/issues/2358.html)	C++14	lambda expressions appearing in default arguments had to be capture-less even if all captures are initialized with expressions which can appear in default arguments	allow such lambda expressions with captures
CWG 2509 (https://cplusplus.github.io/CWG/issues/2509.html)	C++17	each specifier could have multiple occurrences in the specifier sequence	each specifier can only appear at most once in the specifier sequence
CWG 2561 (https://cplusplus.github.io/CWG/issues/2561.html)	C++23	a lambdas with explicit object parameter could have a conversion function to an undesired function pointer type	it does not have such a conversion funtion
CWG 2881 (https://cplusplus.github.io/CWG/issues/2881.html)	C++23	`operator()` with explicit parameter could be instantiated for a derived class when the inheritance was not public or ambiguous	made ill-formed
P0588R1 (https://wg21.link/P0588R1)	C++11	the rule for implicit lambda capture detected odr-use	the detection is simplified

See also

`auto` specifier	specifies a type deduced from an expression
`function`	copyable wrapper of any copy constructible callable object (class template)

External links

Nested function - a function which is defined within another (*enclosing*) function.

Retrieved from "https://en.cppreference.com/mwiki/index.php?title=cpp/language/lambda&oldid=175218"