

Chapter 1

Safe Features

`ch-safe`
`sec-safe-cpp11` Intro text should be here.

Chapter 1 Safe Features

sec-safe-cpp14

Chapter 2

Conditionally Safe Features

ch-conditional

sec-conditional-cpp11

Intro text should be here.

Unnamed Local Function Objects (Closures)

lambda-expression

Lambda expressions provide a means of defining function objects at the point where they are needed, enabling a powerful and convenient way to specify callbacks or local functions.

description

Description

Generic, object-oriented, and functional programming paradigms all place great importance on the ability of a programmer to specify a *callback* that is passed as an argument to a function. For example, the Standard-Library algorithm, `std::sort`, accepts a callback argument specifying the sort order:

```
#include <algorithm>    // std::sort
#include <functional>    // std::greater
#include <vector>        // std::vector

template <typename T>
void sortAscending(std::vector<T>& v)
{
    std::sort(v.begin(), v.end(), std::greater<T>());
}
```

The function object, `std::greater<T>()`, is callable with two arguments of type `T` and returns **true** if the first is greater than the second and **false** otherwise. The Standard Library provides a small number of similar functor types, but, for more complicated cases, the programmer must write a functor themselves. If a container holds a sequence of `Employee` records, for example, we might want to sort the container by name or by salary:

```
#include <string>    // std::string
#include <vector>    // std::vector

struct Employee
{
    std::string name;
    long salary;    // in whole dollars
};

void sortByName(std::vector<Employee>& employees);
void sortBySalary(std::vector<Employee>& employees);
```

The implementation of `sortByName` can delegate the sorting task to the standard algorithm, `std::sort`. However, because `Employee` does not supply `operator<` and to achieve the correct sorting criteria, we will need to supply `std::sort` with a callback that compares the names of two `Employee` objects. We implement this callback as a pointer to a simple function that we pass to `std::sort`:

```
#include <algorithm>    // std::sort

bool nameLt(const Employee& e1, const Employee& e2)
```

C++11

Lambdas

```
// returns true if e1.name is less than e2.name
{
    return e1.name < e2.name;
}

void sortByName(std::vector<Employee>& employees)
{
    std::sort(employees.begin(), employees.end(), &nameLt);
}
```

The `sortBySalary` function can similarly delegate to `std::sort`. For illustrative purposes, we will use a **function object** (a.k.a., **functor**) rather than a function pointer as the callback to compare the salaries of two `Employee` objects. Every **functor class** must provide a **call operator** (i.e., **operator()**), which, in this case, compares the salary fields of its arguments:

```
struct SalaryLt
{
    // Functor whose call operator compares two Employee objects and returns
    // true if the first has a lower salary than the second, false otherwise.

    bool operator()(const Employee& e1, const Employee& e2) const
    {
        return e1.salary < e2.salary;
    }
};

void sortBySalary(std::vector<Employee>& employees)
{
    std::sort(employees.begin(), employees.end(), SalaryLt());
}
```

Although it is a bit more verbose, a call through the **function object** is easier for the compiler to analyze and automatically inline within `std::sort` than is a call through the function pointer. **Function objects** are also more flexible because they can carry state, as we’ll see shortly. The sorting example illustrates how small bits of a function’s logic must be factored out into special-purpose auxiliary functions and/or functor classes that are often not re-usable. It is possible, for example, that the `nameLt` function and `SalaryLt` class are not used anywhere else in the program.

When callbacks are tuned to the specific context in which they are used, they become both more complicated and less re-usable. Let’s say, for example, that we wish to count the number of employees whose salary is above the average for the collection. Using Standard Library algorithms, this task seems trivial: (1) sum all of the salaries using `std::accumulate`, (2) calculate the average salary by dividing this sum by the total number of employees, and (3) count the number of employees with above-average salaries using `std::count_if`. Unfortunately, both `std::accumulate` and `std::count_if` require callbacks to return the salary for an `Employee` and to supply the criterion for counting, respectively. The callback for `std::accumulate` must take two parameters — the current running sum and an element from the sequence being summed — and must return the new running sum:

```
struct SalaryAccumulator
{
    long operator()(long currSum, const Employee& e) const
        // returns the sum of currSum and the salary field of e
    {
        return currSum + e.salary;
    }
};
```

The callback for `std::count_if` is a **predicate** (i.e., an expression that yields a Boolean result in response to a yes-or-no question) that takes a single argument and returns **true** if an element of that value should be counted and **false** otherwise. In this case, we are concerned with `Employee` object’s having salaries above the average. Our **predicate functor** must, therefore, carry around that average so that it can compare it to the salary of the employee that is presented as an argument:

```
class SalaryIsGreater // function object constructed with a reference salary
{
    const long d_referenceSalary;

public:
    explicit SalaryIsGreater(long rs) : d_referenceSalary(rs) { }
        // construct with a reference salary, rs

    bool operator()(const Employee& e) const
        // return true if the salary for Employee e is greater than the
        // reference salary specified on construction, false otherwise
    {
        return e.salary > d_referenceSalary;
    }
};
```

Note that, unlike our previous **functor classes**, `SalaryIsGreater` has a member variable, i.e., it has *state*. This member variable must be initialized, necessitating a constructor. Its **call operator** compares its input argument against this member variable to compute the **predicate** value.

With these two **functor classes** defined, we can finally implement the simple three-step algorithm for determining the number of employees with salaries greater than the average:

```
#include <algorithm> // std::count_if
#include <numeric>   // std::accumulate

std::size_t numAboveAverageSalaries(const std::vector<Employee>& employees)
{
    const long sum = std::accumulate(employees.begin(), employees.end(), 0L,
                                     SalaryAccumulator());

    const long average = sum / employees.size();
    return std::count_if(employees.begin(), employees.end(),
                        SalaryIsGreater(average));
}
```


C++11

Lambdas

}

The first statement creates an object of the `SalaryAccumulator` class and passes that object to the `std::accumulate` algorithm to produce the sum of all of the salaries. The second statement divides the sum by the size of the `employees` collection to compute the average salary. The third statement creates an object of the `SalaryIsGreater` class and passes it to the `std::count_if` algorithm to compute the result. Note that the local variable, `average`, is used to initialize the reference value in the `SalaryIsGreater` object.

We now turn our attention to a syntax that allows us to rewrite these examples much more simply and compactly. Returning to the sorting example, the rewrite has the name-comparison and salary-comparison operations expressed in-place, within the call to `std::sort`:

```
void sortByName2(std::vector<Employee>& employees)
{
    std::sort(employees.begin(), employees.end(),
              [](const Employee &e1, const Employee &e2)
              {
                  return e1.name < e2.name;
              });
}

void sortBySalary2(std::vector<Employee>& employees)
{
    std::sort(employees.begin(), employees.end(),
              [](const Employee &e1, const Employee &e2)
              {
                  return e1.salary < e2.salary;
              });
}
```

In each case, the third argument to `std::sort` — beginning with `[]` and ending with the nearest closing `}` — is called a **lambda expression**. Intuitively, for this case, one can think of a **lambda expression** as an *operation* that can be invoked as a callback by the algorithm. The example shows a function-style parameter list — matching that expected by the `std::sort` algorithm — and a function-like body that computes the needed predicate. Using **lambda expressions**, a developer can express a desired operation directly at the point of use rather than defining it elsewhere in the program.

The compactness and simplicity afforded by the use of **lambda expressions** is even more evident when we rewrite the average-salaries example:

```
std::size_t numAboveAverageSalaries2(const std::vector<Employee>& employees)
{
    const long sum = std::accumulate(employees.begin(), employees.end(), 0L,
                                     [](long currSum, const Employee& e)
                                     {
                                         return currSum + e.salary;
                                     });

    const long average = sum / employees.size();
    return std::count_if(employees.begin(), employees.end(),
```

```

[average](const Employee& e)
{
    return e.salary > average;
});
    
```

The first **lambda expression**, above, specifies the operation for adding another salary to a running sum. The second **lambda expression** returns true if the **Employee** argument, **e**, has a salary that is larger than **average**, which is a local variable *captured* by the **lambda expression**. A **lambda capture** is a set of local variables that are usable within the body of the **lambda expression**, effectively making the **lambda expression** an extension of the immediate environment. We will look at the syntax and semantics of **lambda captures** in more detail in Parts of a Lambda Expression.

Note that the lambda expressions replaced a significant portion of code that was previously expressed as separate functions or **functor** classes. The fact that some of that code reduction is in the form of documentation (comments) increases the appeal of **lambda expressions** to a surprising degree. Creating a named entity such as a function or class imposes on the developer the responsibility to give that entity a meaningful name and sufficient documentation for a future human reader to understand its *abstract* purpose, outside the context of its use, even for one-off, non-reusable entities. Conversely, when an entity is defined right at the point of use, it might not need a name at all, and it is often self-documenting, as in both the sorting and average-salaries examples above. Both the original creation and maintenance of the code is simplified.

Parts of a lambda expression

f-a-lambda-expression

A **lambda expression** has a number of parts and subparts, many of which are optional. For exposition purposes, let’s look at a sample **lambda expression** that contains all of the parts:

```

      introducer
      ^
    /      \
  [&, cap2, cap3](T1 arg1, T2 arg2) mutable noexcept -> R { /* ... */ }
    \      /      \      /      \      /      \
      v              v              v
    capture      declarator      body
    
```

Evaluating a lambda expression creates a temporary **closure** object of an unnamed type called the **closure type**. Each part of a **lambda expression** is described in detail in the subsections below.

closures

Closures

A **lambda expression** looks a lot like an unnamed function definition, and it is often convenient to think of it that way, but a lambda expression is actually more complex than that. First and foremost, a **lambda expression**, as the name implies, is an *expression* rather than a *definition*. The result of evaluating a **lambda expression** is a special **function object**

called a **closure**¹; it is not until the **closure** is *invoked* — which can happen immediately but usually occurs later (e.g., as a callback) — that the actual body of the **lambda expression** gets evaluated.

Evaluating a **lambda expression** creates a temporary **closure object** of an unnamed type called the **closure type**. The **closure type** encapsulates captured variables (see Section 2.2:“??” on page ??) and has a **call operator** that executes the body of the **lambda expression**. Each **lambda expression** has a unique **closure type**, even if it is identical to another **lambda expression** in the program. If the **lambda expression** appears within a template, the **closure type** for each instantiation of that template is unique. Note, however, that, although the **closure object** is an unnamed temporary object, it can be saved in a named variable whose type can be queried. **Closure types** are copy constructible and move constructible, but they have no other constructors and have deleted assignment operators.² Interestingly, it is possible to *inherit* from a **closure type**, provided the derived class constructs its **closure type** base class using only the default or move constructors. This ability to derive from a **closure type** is convenient when implementing certain library features such as `std::bind`, which take advantage of the empty-base optimization:

```
#include <utility> // std::move

template <typename Func>
int callFunc(const Func& f) { return f(); }

void f1()
{
    int i = 5;
    auto c1 = [i]{ return 2 * i; }; // OK, deduced type for c1
    using C1t = decltype(c1);       // OK, named alias for unnamed type
    C1t c1b = c1;                   // OK, copy of c1
    auto c2 = [i]{ return 2 * i; }; // OK, identical lambda expression
    using C2t = decltype(c2);
    C1t c2b = c2;                   // Error, different types, C1t & C2t
    using C3t = decltype([i]{/* ... */}); // Error, lambda expr within decltype

    class C1Derived : public C1t    // OK, inherit from closure type
    {
        int d_auxValue;

    public:
        C1Derived(C1t c1, int aux) : C1t(std::move(c1)), d_auxValue(aux) { }
        int aux() const { return d_auxValue; }
    };

    int ret = callFunc([i]{ return 2 * i; }); // OK, deduced arg type, Func
```

¹The terms *lambda* and *closure* are borrowed from *Lambda Calculus*, a computational system developed by Alonzo Church in the 1930s. Many computer languages have features inspired by Lambda Calculus, although most (including C++) take some liberties with the terminology. See ? and ?.

²C++17 provides default constructors for empty-capture lambdas. Empty-capture lambdas are assignable in C++20.

```
c1b = c1; // Error, assignment of closures is not allowed.
}
```

The types of `c1` and `c2`, above, are different, even though they are token-for-token identical. As there is no way to explicitly name a **closure type**, we use **auto** (in the case of `c1` and `c2` in `f1`) or template-argument deduction (in the case of `f` in `callFunc`) to create variables directly from the **lambda expression**, and we use **decltype** to create aliases to the types of existing closure variables (`C1t` and `C2t`). Note that using **decltype** directly on a **lambda expression** is ill formed, as shown with `C3t`, because there would be no way to construct an object of the resulting unique type. The derived class, `C1Derived`, uses the type alias `C1t` to refer to its base class. Note that its constructor forwards its first argument to the base-class move constructor.

There is no way to specify a **closure type** prior to creating an actual **closure object** of that type. Consequently, there is no way to declare `callFunc` with a parameter of the actual **closure type** that will be passed; hence, it is declared as a template parameter. As a special case, however, if the **lambda capture** is *empty* (i.e., the **lambda expression** begins with `[]`; see Section 2.2:“??” on page ??), then the **closure** is implicitly convertible to an ordinary function pointer having the same signature as its **call operator**:

```
char callFunc2(char (*f)(const char*)) { return f("x"); } // not a template

char c = callFunc2([](const char* s) { return s ? s[0] : '\0'; });
// OK, closure argument is converted to function-pointer parameter

char d = callFunc2([c](const char* s) { /* ... */ });
// Error, lambda capture is not empty; no conversion to function pointer
```

The `callFunc2` function takes a callback in the form of a pointer to function. Even though it is not a template, it can be called with a lambda argument having the same parameter types, the same return type, and an empty **lambda capture**; the **closure object** is converted to an ordinary pointer to function. This conversion is *not* available in the second call to `callFunc2` because the **lambda capture** is not empty.

Conversion to function pointer is considered a user-defined conversion operator and thus cannot be implicitly combined with other conversions on the same expression. It can, however, be invoked *explicitly*, as needed:

```
using Fp2 = int (*)(int); // function-pointer type

struct FuncWrapper
{
    FuncWrapper(Fp2) { /* ... */ } // implicit conversion from function-pointer
    // ...
};

int f2(FuncWrapper) { /* ... */ return 0; }
int i2 = f2([](int x) { return x; }); // Error, two user-defined conversions
int i3 = f2(static_cast<Fp2>([](int x) { return x; })); // OK, explicit cast
int i4 = f2(+[](int x) { return x; }); // OK, forced conversion
```

The first call to `f2` fails because it would require two implicit user-defined conversions: one from the **closure type** to the `Fp2` function-pointer type and one from `Fp2` to `FuncWrapper`. The second call succeeds because the first conversion is made explicit with the `static_cast`. The third call is an interesting shortcut that takes advantage of the fact that unary `operator+` is defined as the identity transformation for pointer types. Thus, the **closure-to-pointer** conversion is invoked for the operand of `operator+`, which returns the unchanged pointer, which, in turn, is converted to `FuncWrapper`; the first and third steps of this sequence use only one user-defined conversion each. The Standard Library `std::function` class template provides another way to pass a function object of unnamed type, one that does not require the **lambda capture** to be empty; see *Use Cases* on page 30.

The compile-time and runtime phases of defining a **closure type** and constructing a **closure object** from a single **lambda expression** resembles the phases of calling a function template; what looks like an ordinary function call is actually broken down into a compile-time instantiation and a runtime call. The **closure type** is deduced when a **lambda expression** is encountered during compilation. When the control flow passes through the **lambda expression** at run time, the **closure object** is *constructed* from the list of captured local variables. In the `numAboveAverageSalaries` example in the Description section, the `SalaryIsGreater` class can be thought of as a **closure type** — created by hand instead of by the compiler — whereas the call to `SalaryIsGreater(average)` is analogous to constructing the **closure object** at run time.

Finally, the purpose of a **closure** is to be invoked. It can be invoked immediately by supplying arguments for each of its parameters:

```
#include <iostream> // std::cout
void f3()
{
    [](const char* s) { std::cout << s; }("hello world\n");
    // equivalent to std::cout << "hello world\n";
}
```

The **closure object**, in this example, is invoked immediately and then destroyed, making the above just a complicated way to say `std::cout << "hello world\n";`. More commonly, the **lambda expression** is used as a local function for convenience and to avoid clutter:

```
#include <cmath> // std::sqrt

double hypotenuse(double a, double b)
{
    auto sqr = [](double x) { return x * x; };
    return std::sqrt(sqr(a) + sqr(b));
}
```

Note that there is no way to overload calls to **closures**:

```
auto sqr = [](int x) { return x * x; }; // OK, store **closure** in sqr
auto sqr = [](double x) { return x * x; }; // Error, redefinition of sqr
```

The most common use of a **lambda expression**, however, is as a callback to a function template, e.g., as a functor argument to an algorithm from the Standard Library:

```
#include <algorithm> // std::partition

template <typename FwdIt>
FwdIt oddEvenPartition(FwdIt first, FwdIt last)
{
    using value_type = decltype(*first);
    return std::partition(first, last, [](value_type v) { return v % 2 != 0; });
}
```

The `oddEvenPartition` function template moves odd values to the start of the sequence and even values to the back. The **closure object** is invoked repeatedly within the `std::partition` algorithm.

Lambda capture and lambda introducer

and-lambda-introducer

The purpose of the **lambda capture** is to make certain local variables from the environment available to be used (or, more precisely, **ODR-used**, which means that they are used in a potentially-evaluated context) within the **lambda body**. Each local variable can be **captured by copy** or **captured by reference**. Orthogonally, each variable can be **explicitly captured** or **implicitly captured**. When a **lambda expression** appears within a non-static member function, the **this** pointer can be captured as a special case. We’ll examine each of these aspects of **lambda capture** in turn. An extension to **lambda capture** in C++14 is discussed in *C++14 init capture* on page 21.

Syntactically, the **lambda capture** consists of an optional **capture default** followed by a comma-separated list of zero or more identifiers (or the keyword **this**), which are **explicitly captured**. The capture default can be one of `=` or `&` for **capture by copy** or **capture by reference**, respectively. If there is a **capture default**, then **this** and any local variables in scope that are **ODR-used** within the **lambda body** and not **explicitly captured** will be **implicitly captured**.

```
void f1()
{
    int a = 0, b = 1, c = 2;
    auto c1 = [a, b]{ return a + b; };
    // a and b are explicitly captured.
    auto c2 = [&]{ return a + b; };
    // a and b are implicitly captured.
    auto c3 = [&, b]{ return a + b; };
    // a is implicitly captured and b is explicitly captured.
    auto c4 = [a]{ return a + b; };
    // Error, b is ODR-used but not captured.
}
```

The Standard defines the **lambda introducer** as the **lambda capture** together with its surrounding `[` and `]`. If the **lambda introducer** is an empty pair of brackets, no variables will be captured.

```
auto c1 = []{ /* ... */ }; // Empty **lambda capture**
```

The **lambda capture** enables access to portions of the local stack frame. As such, only variables with *automatic storage duration* — i.e., non-static local variables — can be captured, as we’ll see in detail later in this section and in **lambda body**. An **explicitly captured** variable whose name is immediately preceded by an **&** symbol in the **lambda capture** is **captured by reference**; without the **&**, it is **captured by copy**. If the **capture default** is **&**, then all **implicitly captured** variables are **captured by reference**. Otherwise, if the **capture default** is **=**, all **implicitly captured** variables are **captured by copy**:

```
void f2a()
{
    int a = 0, b = 1;
    auto c1 = [&a]{ /* ... */ return a; }; // a captured by reference
    auto c2 = [a]{ /* ... */ return a; }; // a captured by copy
    auto c3 = [a, &b]{ return a + b; };
        // a is explicitly captured by copy and b is explicitly
        // captured by reference.
    auto c4 = [=]{ return a + b; };
        // a and b are implicitly captured by copy.
    auto c5 = [&]{ return &a; };
        // a is implicitly captured by reference.
    auto c6 = [&, b]{ return a * b; };
        // a is implicitly captured by reference and b is explicitly
        // captured by copy.
    auto c7 = [=, &b]{ return a * b; };
        // a is implicitly captured by copy and b is explicitly
        // captured by reference.
    auto c8 = [a]{ return a * b; };
        // Error, a is explicitly captured by copy, but b is not captured.
    auto c9 = [this]{ /* ... */ }; // Error, no this in nonmember function
}

class Class1a
{
public:
    void mf1()
    {
        auto c12 = [this]{ return this; }; // Explicitly capture this.
        auto c13 = [=] { return this; }; // Implicitly capture this.
    }
};
```

Redundant captures are not allowed; the same name (or **this**) cannot appear twice in the **lambda capture**. Moreover, if the **capture default** is **&**, then none of the **explicitly captured** variables may be **captured by reference**, and if the **capture default** is **=**, then any **explicitly captured** entities can be neither **explicitly copied** variables nor **this**³:

```
class Class1b
{
```

³C++20 removed the prohibition on explicit capture of **this** with an **=** capture default.

```
public:
    void mf1()
    {
        int a = 0;
        auto c1 = [a, &a]{ /* ... */ }; // Error, a is captured twice.
        auto c2 = [=, a]{ /* ... */ };
            // Error, explicit capture of a by copy is redundant.
        auto c3 = [&,&a]{ /* ... */ };
            // Error, explicit capture of a by reference is redundant.
        auto c4 = [=, this]{ return this; };
            // Error, explicit capture of this with = capture default
    }
};
```

We’ll use the term *primary variable* to refer to the block-scope local variable outside of the **lambda expression** and *captured variable* to refer to the variable of the same name as viewed from within the **lambda body**. For every object that is **captured by copy**, the **lambda closure** will contain a member variable having the same name and type, after stripping any reference qualifier (except reference-to-function); this member variable is initialized from the primary variable by direct initialization and is destroyed when the **closure object** is destroyed. Any **ODR-use** of that name within the **lambda body** will refer to the **closure’s** member variable. Thus, for an entity that is **captured by copy**, the primary and captured variables refer to distinct objects with distinct lifetimes. By default, the **call operator** is **const**, providing read-only access to members of the **closure object** (i.e., captured variables that are **captured by copy**); mutable (non**const**) **call operators** are discussed in *Lambda declarator* on page 22:

```
assert
```

```
void f3()
{
    int a = 5;
    auto c1 = [a]           // a is captured by copy.
    {
        return a;          // return value of copy of a
    };
    a = 10;                 // Modify a after it was captured by c1.
    assert(5 == c1());      // OK, a within c1 had value from before the change.

    int& b = a;
    auto c2 = [b]           // b is int (not int&) **captured by copy**.
    {
        return b;          // return value of copy of b
    };
    b = 15;                 // Modify a through reference b.
    assert(10 == c2());     // OK, b within c2 is a copy, not a reference.

    auto c3 = [a]
    {
```


C++11

Lambdas

```
        ++a;           // Error, a is const within the lambda body.
    };
}
```

In the example above, the **lambda expression** is evaluated to produce a **closure object**, **c1**, that captures a *copy* of **a**. Even when the primary **a** is subsequently modified, the captured **a** in **c1** remains unchanged. When **c1** is invoked, the **lambda body** returns the *copy*, which still has the value 5. The same applies to **c2**, but note that the copy of **b** is *not a reference* even though **b** is a reference. Thus, the copy of **b** in **c2** is the value of **a** that **b** referred to at the time that **c2** was created.

When a variable is **captured by reference**, the captured variable is simply an alias to the primary variable; no copies are made. It is, therefore, possible to modify the primary variable and/or take its address within the **lambda body**:

```
assert
void f4()
{
    int a = 5;
    auto c1 = [&a]    // a is **captured by reference**.
    {
        a = 10;      // Modify a through the captured variable.
        return &a;    // return address of captured a
    };
    assert(c1() == &a); // OK, primary and captured a have the same address.
    assert(10 == a);   // OK, primary a is now 10.

    int& b = a;
    auto c2 = [&b]    // b is **captured by reference**
    {
        return &b;    // return address of captured b
    };
    assert(c2() == &b); // OK, primary and captured b have the same address.
    assert(c2() == &a); // OK, captured b is an alias for a.
}
```

In contrast to the **f3** example, the **c1 closure object** above does *not* hold a copy of the captured variable, **a**, though the compiler may choose to define a member of type **int&** that refers to **a**. Within the **lambda body**, modifying **a** modifies the primary variable, and taking its address returns the address of the primary variable, i.e., the captured variable is an alias for the primary variable. With respect to variables that are **captured by reference**, the **lambda body** behaves very much as though it were part of the surrounding block. The lifetime of a variable that is **captured by reference** is the same as that of the primary variable (since they are the same). In particular, if a copy of the **closure object** outlives the primary variable, then the captured variable becomes a *dangling reference*; see *Potential Pitfalls* on page 38.

If **this** appears in the **lambda capture**, then (1) the current **this** pointer is **captured by copy** and (2) within the **lambda body**, member variables accessible through **this** can be used without prefixing them with **this->**, as though the **lambda body** were an extension of the surrounding member function. The **lambda body** cannot refer to the **closure** directly;

the captured **this** does not point to the **closure** but to the ***this** object of the function within which it is defined:

```
assert

struct Class1
{
    int d_value;

public:
    // ...
    void mf() const
    {
        auto c1 = []{ return *this; };           // Error, this is not captured.
        auto c2 = []{ return d_value; };         // Error, this is not captured.
        auto c3 = [d_value]{ /* ... */; };       // Error, cannot capture member
        auto c4 = [this]{ return this; };        // OK, returns this
        auto c5 = [this]{ return d_value; };     // OK, returns this->d_value
        assert(this == c4());                   // OK, captured this is Class1.
    }
};
```

Note that **c4** returns **this**, which is the address of the **Class1** for which **mf** was called. This is one way in which the **closure type** is different from a named **functor type** — there is no way for an object of **closure type** to refer to itself directly. Because the **closure type** is unnamed and because it does not supply its own **this** pointer, it is difficult (but not impossible) to create a *recursive lambda expression*; see **Usage examples [there is no section with this name]** .

If **this** is captured (implicitly or explicitly), the **lambda body** will behave much like an extension of the member function in which the **lambda expression** appears, with direct access to the class’s members:

```
std::count_ifstd::vector

class Class2
{
    int d_value;

public:
    std::size_t mem(const std::vector<int>& v) const
    {
        auto f = []{ return d_value; };
        // Error, this not captured; can't see d_value.
        return std::count_if(v.begin(), v.end(),
                             [this](int element){ return element < d_value; });
        // OK, uses this->d_value.
    }
};
```

Note that capturing **this** does not copy the class object that it points to; the original **this** and the captured **this** will point to the same object:

C++11

Lambdas

```

    assert

class Class3
{
    int d_value;

public:
    void mf()
    {
        auto c1 = [this]{ ++d_value; }; // increment this->d_value
        d_value = 1;
        c1();
        assert(2 == d_value);          // change to d_value is visible
    }
};

```

Here, we captured **this** in **c1** but then proceeded to modify the object pointed to by **this** within the **lambda body**.⁴

A **lambda expression** can occur wherever other expressions can occur, including within other **lambda expressions**. The set of entities that can be captured in a valid **lambda expression** depends on the surrounding scope. A **lambda expression** that does not occur immediately within **block scope** cannot have a **lambda capture**:

```

namespace ns1
{
    int v = 10;
    int w = [v]{ /* ... */ return 0; }();
    // Error, capture in global/namespace scope

    void f4(int a = [v]{ return v; }()); // Error, capture in default argument
}

```

When a **lambda expression** occurs in block scope, it can capture any local variables with *automatic* (i.e., non-static) storage duration in its **reaching scope**. The Standard defines the **reaching scope** of the **lambda expression** as the set of enclosing scopes up to and including the innermost enclosing function and its parameters. Static variables can be used without capturing them; see *Lambda body* on page 27:

```

void f5(const int& a)
{
    int b = 2 * a;
    if (a)
    {
        int c;
        // ...
    }
    else
    {

```

⁴In C++17, it is possible to capture ***this**, which results in the entire class object being copied, not just the **this** pointer.

```

        int d = 4 * a;
    static int e = 10;

    auto c1 = [a]{ /* ... */ };    // OK, capture argument a from f5.
    auto c2 = [=]{ return b; };    // OK, implicitly capture local b.
    auto c3 = [&c]{ /* ... */ };    // Error, c is not in **reaching scope**.
    auto c4 = [&]{ d += 2; };    // OK, implicitly capture local d.
    auto c5 = [e]{ /* ... */ };    // Error, e has static duration.
}

struct LocalClass
{
    void mf()
    {
        auto c6 = [b]{ /* ... */ };    // Error, b not in **reaching scope**
    }
};

```

The **reaching scope** of the **lambda expressions** for **c1** through **c5**, above, includes the local variable **d** in the **else** block, **b** in the surrounding function block, and **a** from **f5**’s arguments. The local variable, **c**, is not in their **reaching scope** and cannot be captured. Although **e** *is* in their **reaching scope**, it cannot be captured because it does not have automatic storage duration. Finally, the **lambda expression** for **c6** is within a member function of a local class. Its **reaching scope** ends with the innermost function, **LocalClass::mf**, and does not include the surrounding block that includes **a** and **b**.

Only when the innermost enclosing function is a non-static class member function can **this** be captured:

```

void f5()
{
    auto c1 = [this]{ /* ... */ };    // Error, f5 is not a member function.
}

class Class3
{
    static void mf1()
    {
        auto c2 = [this]{ /* ... */ };    // Error, mf1 is static.
    }

    void mf2()
    {
        auto c3 = [this]{ /* ... */ };    // OK, mf2 is non-static member function

        struct LocalClass
        {
            static void mf3()
            {

```

C++11

Lambdas

```

        auto c4 = [this]{ /* ... */ };
        // Error, innermost function, mf3, is static
    }
};
}
};

```

When a **lambda expression** is enclosed within another **lambda expression**, then the **reaching scope** includes all of the intervening **lambda bodies**. Any variable captured (implicitly or explicitly) by the inner **lambda expression** must be either defined or captured by the enclosing lambda expression:

```

void f6()
{
    int a, b, c;
    const char* d;
    auto c1 = [&a]                // capture a from function block
    {
        int d;                    // local definition of d hides outer def
        auto c2 = [&a]{ /* ... */ }; // OK, a is captured in enclosing lambda
        auto c3 = [d]{ /* ... */ }; // OK, capture int d from enclosing
        auto c4 = [&]{ return d; }; // OK, " " " "
        auto c5 = [b]{ /* ... */ }; // Error, b is not captured in enclosing
    };
    auto c6 = [=]
    {
        auto c7 = [&]{ return b; };
        // OK, ODR-use of b causes implicit capture in c7 and c6.
        auto c8 = [&d]{ return &d; };
        // d is captured by copy in c6; c8 returns address of copy
    };
}

```

Note that there are two variables named **d**: one at function scope and one within the body of the first **lambda expression**. Following normal rules for unqualified name lookup, the inner **lambda expressions** used to initialize **c3** and **c4** capture the *inner* **d** (of type **int**), not the *outer* **d** (of type **const char***). Because it is not captured, primary variable **b** is *visible* but not *usable* — an important distinction that we’ll discuss in *Lambda body* on page 27 — within the body of **c1** and cannot, therefore, be captured by **c5**.

The **lambda body** for **c7** **ODR-uses** **b**, thus causing it to be **implicitly captured**. This capture by **c7** constitutes an **ODR-use** of **b** within the enclosing **lambda expression**, **c6**, in turn causing **b** to be **implicitly captured** by **c6**. In this way, a single **ODR-use** can trigger a *chain* of **implicit captures** from an enclosed **lambda expression** through its enclosing **lambda expressions**. Critically, when a variable is **captured by copy** in one **lambda expression**, any enclosed **lambda expressions** that capture the same name will capture the *copy*, not the primary variable, as we see in the **lambda expression** for **c8**.

Note that, when a variable is named in a **lambda capture**, it isn’t automatically *captured*. A variable is not captured unless it is **ODR-used** within the **lambda expression**:

```

void f7()

```

```
{
    int      a = 0;
    int const b = 2;
    auto c1 = [a]{ return 2 * a; };      // OK, a is explicitly captured.
    auto c2 = [a]{ return 0; };          // Warning, no uses of a
    auto c3 = [a]{ return sizeof(a); };  // Warning, sizeof(a) isn't an ODR-use
    auto c4 = [b]{ return b; };          // Warning, value of b isn't an ODR-use
    auto c5 = [&b]{ return &b; };        // OK, address of b is an ODR-use
}
```

In the above example, the **lambda body** for **c1** contains an **ODR-use** of **a** and thus captures **a**. Conversely, **c2** does *not* capture **a** because the **lambda body** for **c2** does not contain an **ODR-use** of **a**; most compilers will issue a warning diagnostic about the superfluous presence of **a** in the **lambda capture** for **c2**. The case of **c3** is a bit more subtle. Although **a** is *used* within **c3**, it is not **ODR-used** — i.e., it is not used in a potentially-evaluated context because **sizeof** does not evaluate its argument — so the warning is the same as for **c2**; see *Potential Pitfalls* on page 38. The last two cases are a bit more subtle: The use of the *value* of a **const** variable (in **c4**) is *not* an **ODR-use** of that variable, whereas the use its *address* (in **c5**) *is* an **ODR-use**.

Finally, a **lambda capture** within a **variadic function template** (see Section 2.1.“??” on page ??) may contain a **parameter pack expansion**:

```
#include <utility> // std::forward

template <typename... ArgTypes>
int f8(const char* s, ArgTypes&&... args);

template <typename... ArgTypes>
int f9(ArgTypes&&... args)
{
    const char* s = "Introduction";
    auto c1 = [=]{ return f8(s, args...); }; // OK, args... captured by copy
    auto c2 = [s,&args...]{ return f8(s, std::forward<ArgTypes>(args)...); };
    // OK, explicit capture of args... by reference
}
```

In the example above, the variadic arguments to **f9** are **implicitly captured** using **capture by copy** in the first **lambda expression**. This means that, regardless of the **value category** (*rvalue*, *lvalue*, and so on) of the original arguments, the captured variables are all *lvalue* members of the resulting *closure*. Conversely, the second **lambda expression** captures the set of arguments using **capture by reference**, again resulting in captured variables that are *lvalues*. The **ArgTypes parameter pack expansion** designates a list of *types*, not *variables*, and does not, therefore, need to be captured to be used within the **lambda expression**, nor would it be valid to attempt to capture it. Because **ArgTypes** is specified using a forwarding reference (**&&**, see Section 2.1.“??” on page ??), the Standard Library function, **std::forward**, can be used to cast the captured variables to the **value category** of their corresponding arguments.

c++14-init-capture

C++14 init capture

TODO VR: this subsection shouldn't be here, we have a specific C++14 feature 'lambdacapture' for this stuff.

Each item in a C++11 **lambda capture** is either a **capture default** or a **simple capture** consisting of the name of a local variable, either by itself (for **capture by copy**) or preceded by an **&** (for **capture by reference**). C++14 introduces another possibility, an **init capture**, consisting of a variable name and an initializer, which creates a new captured variable initialized to an arbitrary expression. The initializer can be either preceded by an **=** token or can be a braced initialization (see Section 2.1.“??” on page ??). The newly defined variable does not necessarily share the name or type of a primary local variable:

```
void f1(int a, bool bits[])
{
    double g(int);

    int b;
    auto c1 = [i{5}]{ /* ... */ };           // Define int i = 5.
    auto c2 = [b=bits[a]]{ /* ... */ };      // Define bool b = copy of bits[a].
    auto c3 = [&r=a, b]{ /* ... */ };         // int& r = reference to a. Copy b.
    auto c4 = [&, p=bits+1]{ *p = !a; };      // bool* p = pointer to bits[1].
    auto c5 = [x=g(a)]{ return x; };         // double x = g(a).
}
```

In the **lambda expression** for **c1**, above, the **init capture** defines a new variable, **i**, initialized with the value 5. Within the **lambda body**, this **i** is indistinguishable from any other variable captured by the **closure**, but, outside of the **lambda body**, it differs from a **simple capture** in that it does not capture a local variable. The **lambda capture** for **c2** similarly defines a new variable, **b**, initialized from an expression involving **bits** and **a**, but does not capture either of them. Once captured, **bits[a]** can change without affecting the value of **b**. An **init capture** can also define a variable of *reference* type, as shown in the **init capture** for **c3**. The **lambda capture** for **c3** also shows an **init capture** mixed in with **simple captures**. Note, however, that the **capture default**, if any, has no effect on the **init capture**, as shown in the **lambda expression** for **c4**. The **init capture** for **c5** shows an arbitrary expression being captured in this the return value of a function call.

The variable defined in an **init capture** is defined as if by the declaration:

```
auto *init-capture* ;
```

The hypothetical variable created by such a definition is unique and separate from any similarly named variable in the environment. This uniqueness allows the same name to appear on both the left and right of the **=** symbol:

```
#include <string>    // std::string
#include <utility>    // std::move

void f2(std::string s)
{
    float a = 1.2;
    auto c1 = [&s=s]{ /* ... */ };           // effectively capture by reference
}
```

```

auto c2 = [a=static_cast<double>(a)]{ /* ... */ };
auto c3 = [s=std::move(s)]{ /* ... */ }; // effectively capture by move
std::string s2 = s;                      // BAD IDEA, s is moved from
    }
```

In all of the **lambda expressions** above, an **init capture** defines a variable whose name shadows one in the environment. In fact, the first use, in **c1**, defines a captured variable, **s**, that is a reference to the argument variable by the same name, yielding a behavior within the **lambda body** that is essentially indistinguishable from normal **capture by reference**. The second **lambda expression** copies the local variable, **a**, into a captured variable such that, within the **lambda body**, the variable **a** is a **double**. Finally, the **lambda capture** for **c3** creates a captured variable, **s**, initialized from the argument **s** using the **std::string move constructor** (see Section 2.1.?? on page ??), simulating something that many considered to be missing in C++11: capture by move. Note, however, that after constructing the **c3 closure**, the local variable, **s**, is in a *moved-from* state and thus has an unspecified value, even if **c3** is never invoked.

In C++14, an **init capture** cannot be a **parameter pack expansion**.⁵

lambda-declarator

Lambda declarator

The **lambda declarator** looks a lot like a function declaration and is effectively the declaration of the **closure type**’s **call operator**. The **lambda declarator** comprises the **call operator**’s parameter list, mutability, exception specification, and return type:

```

[/* ... */](T1 arg1, T2 arg2) mutable noexcept(/* ... */) -> RetType { /* ... */ }
```

Although the **lambda declarator** looks very similar to a function declaration, we cannot forward declare any part of a **lambda expression**; we can only define it.

The entire **lambda declarator** is optional. However, if *any* part is present, the parameter list must be present (even if it declares no parameters):

```

auto c1 = [](int x) noexcept { /* ... */ }; // OK, param list and exception spec
```

⁵A syntax for **parameter pack expansion** of **init captures** was introduced in C++20:

```

template <typename T> class X { /* ... */ };
template <typename T> X<T> f3(T&&);

template <typename... ArgTypes>
void f4(ArgTypes&&... args)
{
    auto c1 = [...x=f3(std::forward<ArgTypes>(args))]{ /* ... */ };
    // Error in C++14. OK in C++20
}
```



```
auto c2 = []() -> int { return 5; };           // OK, with ret type empty param list
auto c3 = [] -> double { /* ... */ };         // Error, ret type with no param list
```

The parameter list for a **lambda expression** is the same as a parameter list for a function declaration, with minor modifications.

1. A parameter is not permitted to have the same name as an explicitly-captured variable:

```
void f1()
{
    int a;
    auto c1 = [a](short* a){ /* ... */ }; // Error, parameter shadows captured a.
    auto c2 = [](short* a){ /* ... */ }; // OK, parameter hides local a.
    auto c3 = [=](short* a){ /* ... */ }; // OK, local a is not captured.
}
```

In the definition of `c1`, the **lambda expression explicitly captures a**, then improperly tries to declare a parameter by the same name. When `a` is not captured, as in the **lambda expression** for `c2`, having a parameter named `a` does not pose a problem; within the **lambda body**, the declaration of `a` in the parameter list will prevent name lookup from finding the declaration in the enclosing scope. The situation with `c3` is essentially the same as for `c2`; because name lookup finds `a` in the parameter list rather than the enclosing scope, it does not attempt to capture it.

2. In C++11, none of the parameters may have default arguments. This restriction does not apply to C++14 and after:

```
auto c4 = [](int x, int y = 0){ /* ... */ }; // Error in C++11. OK in C++14.
```

3. If the type of any of the parameters contains the keyword **auto**, then the **lambda expression** becomes a **generic lambda**; see Section 2.2:“??” on page ?? . Everything in this chapter applies to **generic lambda** as well as regular **lambda expressions**, so it is recommended that you read the rest of this chapter before moving on to the **generic lambdas** feature.

Note that, unlike the **lambda capture**, the parameter list is usually dictated by the *client* of a **lambda expression** rather than by its author. Moreover, the **lambda capture** is evaluated only once when the **closure** is created, whereas the parameter list is bound to actual arguments each time the **call operator** is invoked:

```
#include <iostream> // std::cout
#include <vector>    // std::vector

template <typename InputIter, typename Func>
void applyToEveryOtherElement(InputIter start, InputIter last, Func f)
    // For elements in the range [start, last), invoke f on the first
    // element, skip the second element, etc., alternating between calling f
    // and skipping elements.
{
    while (start != last)
```

```

    {
        f(*start++);                // Process one element.
        if (start != last) { start++; } // Skip one element.
    }
}

void f2(const std::vector<float>& vec)
{
    std::size_t size = vec.size();
    applyToEveryOtherElement(vec.begin(), vec.end(),
        [](float x){ /* ... */ }); // OK, one float parameter
    applyToEveryOtherElement(vec.begin(), vec.end(),
        [](float x, int y){ /* ... */ }); // Error, too many parameters
    applyToEveryOtherElement(vec.begin(), vec.end(),
        [size](){ /* ... */ }); // Error, too few parameters
    applyToEveryOtherElement(vec.begin(), vec.end(),
        [size](double x){ /* ... */ }); // OK, convertible from float
}

```

In the above definition of `applyToEveryOtherElement`, the callable argument, `f`, is applied to half of the elements of the input range and has the **closure type** resulting from a **lambda expression**. In this example, therefore, each instantiation of `applyToEveryOtherElement` calls its `f` parameter multiple times with a single argument of type `float`. In the first call to `applyToEveryOtherElement`, the **lambda closure** has a parameter list consisting of a single parameter of type `float` and is thus compatible with the expected signature of `f`. The second and third calls to `applyToEveryOtherElement` supply a **lambda closure** with too many and too few parameters, respectively, resulting in a compilation error at the point where `f` is called. The last call to `applyToEveryOtherElement` supplies a **lambda closure** that takes a single argument of type `double`. Since an argument of type `float` is convertible to `double`, this **lambda expression** is also a valid argument to `applyToEveryOtherElement`. Note that the presence or absence of a **lambda closure** makes no difference to the validity of the **lambda expression** from the point of view of the client function.

The **mutable** keyword, if present, indicates that the **call operator** for the **closure type** should *not* be **const**. Recall that a normal class member function can modify the class’s members if not declared **const**. The **call operator** is just an ordinary member function in this regard:

```

class Class1
{
    int d_value;

public:
    // ...
    void operator()(int v)      { d_value = v; } // OK, object is mutable
    void operator()(int v) const { d_value = v; } // Error, object is const
};

```

The **const** version of `operator()` cannot modify the member variables of the `Class1` object, whereas the undecorated one can. The **call operator** for a **closure type** has the inverse

default **constness**: the **call operator** is implicitly **const** *unless* the **lambda declarator** is decorated with **mutable**. In practice, this rule means that member variables of the **closure object**, i.e., variables that were **captured by copy**, are **const** by default and cannot be modified within the **lambda body** unless the **mutable** keyword is present:

```
assert
void f2()
{
    int a = 5;
    auto c1 = [a]()          { return ++a; }; // Error, copy of a is const
    auto c2 = [a]() mutable { return ++a; }; // OK, increment *copy* of a
    assert(6 == c2());        // OK, captured a incremented
    assert(5 == a);           // OK, primary a not changed
    assert(7 == c2());        // OK, captured a incremented
}
```

The two **lambda expressions** are identical except for the **mutable** keyword. Both use **capture by copy** to capture local variable **a**, and both try to increment **a**, but only the one decorated with **mutable** can perform that modification. When the **call operator** on **closure object** **c2** is invoked, it increments the *captured copy* of **a**, leaving the primary **a** untouched. If **c2()** is invoked again, it increments its copy of **a** a second time. Using **capture by reference** allows the primary variable to be changed within the **lambda body** regardless of the presence or absence of the **mutable** keyword. Similarly, member variables accessed through **this** are unaffected by the **mutable** keyword, but they *are* affected by the **constness** of the surrounding member function:

```
assert
class Class2
{
    int d_value;
public:
    // ...
    void mf()
    {
        d_value = 1;
        int a = 0;
        auto c1 = [&a, this]{
            a = d_value; // OK, a is a reference to a non-const object.
            d_value *= 2; // OK, this points to a non-const object.
        };
        c1();
        assert(1 == a && 2 == d_value); // values updated by c1
        c1();
        assert(2 == a && 4 == d_value); // values updated by c1 again
    }

    void cmf() const
    {
        int a = 0;
        auto c2 = [=]() mutable {
```

```

    ++a;          // OK, increment mutable **captured variable**
    ++d_value;    // Error, *this is const within cmf.
};
}
};

```

The **lambda expression** for `c1` is not decorated with **mutable** yet both `a` and `d_value` can be modified; the first because it was **captured by reference**, and the second because it was accessed through **this** within a **nonconst** member function, `mf`. Conversely, the **lambda expression** for `c1` cannot modify `d_value` even though it is declared **mutable** because the captured **this** pointer points to a **const** object within the surrounding member function, `cmf`.

The **lambda declarator** may include an **exception specification**, consisting of a **throw** or **noexcept** clause, after the **mutable** decoration, if any. The syntax and meaning of the **exception specification** is identical to that of a normal function; see Section 3.1.“??” on page ??.

The return type of the **call operator** can be determined either by a **trailing return type** or by a **deduced return type**. Every example we have seen up to this point has used a **deduced return type** whereby the return type of the **closure’s call operator** is deduced by the type of object returned by the return statement(s). If there are no return statements in the **lambda body** or if the return statements have no operands, then the return type is **void**. If there are multiple return statements, they must agree with respect to the return type. The rules for a **deduced return type** are the same for a **lambda expression** as they are in C++14 for an ordinary function; see Section 3.2.“??” on page ??:

```

void f3()
{
    auto c1 = [](int& i){ i = 0; }; // deduced return type is void
    auto c2 = []{ return "hello"; }; // deduced return type is const char*
    auto c3 = [](bool c)           // deduced return type is int
    {
        if (c) { return 5; }
        else { return 6; }
    };
    auto c4 = [](bool c)
    {
        if (c) { return 5; } // deduced return type is int
        else { return 6.0; } // Error, double does not match int.
    };
};
}

```

All four of the above **lambda expressions** have a **deduced return type**. The first one deduces a return type of **void** because the **lambda body** has no **return** statements. The next one deduces a return type of **const char*** because the string literal, `"hello"`, decays to a **const char*** in that context. The third one deduces a return type of **int** because all of the **return** statements return values of type **int**. The last one fails to compile because the two branches return values of different types.⁶

⁶The original C++11 Standard did not allow a **deduced return type** for a **lambda body** containing

If a **deduced return type** is impossible or undesirable (see Section 3.2.??” on page ?? for a description of why this feature needs to be used with care), a **trailing return type** can be specified (see Section 1.1.??” on page ??:

```
std::pair
void f4()
{
    auto c1 = [](bool c) -> double {
        if (c) { return 5; }      // OK, int value converted to double
        else { return 6.0; }     // OK, double return value
    };
    auto c2 = []() -> std::pair<int, int> {
        return { 5, 6 };        // OK, brace-initialize returned pair
    };
}
```

In the first **lambda expression** above, we specify a **trailing return type** of **double**. The two branches of the **if** statement would return different types (**int** and **double**), but, because the return type has been definitively declared, the compiler converts the return values to the known return type (**double**). The second **lambda expression** returns a value by brace initialization, which is insufficient for deducing a return value. Again, the ambiguity is resolved by declaring the return value explicitly. Note that, unlike ordinary functions, a **lambda expression** cannot have a return type specified before the **lambda introducer** or **lambda declarator**:

```
auto c5 = int [](int x){ return 0; }; // Error, return type misplaced
auto c6 = [] int (int x){ return 0; }; // Error, return type misplaced
auto c7 = [](int x) -> int{ return 0; }; // OK, trailing return type
```

Attributes (see Section 1.1.??” on page ??) that appertain to the *type* of **call operator** can be inserted in the **lambda declarator** just before the **trailing return type**. If there is no **trailing return type**, the attributes can be inserted before the open brace of the **lambda body**. Unfortunately, these attributes do not apply to the **call operator** itself, but to its type, ruling out some common attributes:

```
#include <cstdlib> // std::abort
auto c1 = []() noexcept [[noreturn]] { // Error, [[noreturn]] on a type
    std::abort();
};
```

Lambda body

lambda-body

Combined, the **lambda declarator** and the **lambda body** make up the declaration and definition of an **inline** class member function that is the **call operator** for the **closure type**. For the purposes of name lookup and the interpretation of **this**, the **lambda body** is considered to be in the context where the **lambda expression** is evaluated (independent of the context where the **closure’s call operator** is invoked).

anything other than a single **return** statement. This restriction was lifted by a defect report and is no longer part of C++11. Compiler versions that predate ratification of this defect report might reject **lambda expressions** having multiple statements and a **deduced return type**.

Critically, the set of entity names that can be used from within the **lambda body** is not limited to captured local variables. Types, functions, templates, constants, and so on — just like any other member function — do not need to be captured and, in fact, *cannot* be captured in most cases. To illustrate, let’s create a number of entities in multiple scopes:

```
#include <iostream> // std::cout

namespace ns1
{
    void f1() { std::cout << "ns1::f1" << '\n'; }
    struct Class1 { Class1() { std::cout << "ns1::Class1()" << '\n'; } };
    int g0 = 0;
}

namespace ns2
{
    void f1() { std::cout << "ns2::f1" << '\n'; }

    template <typename T>
    struct Class1 { Class1() { std::cout << "ns2::Class1()" << '\n'; } };

    int const g1 = 1;
    int      g2 = 2;

    class Class2
    {
        int      d_value; // non-static member variable
        static int s_mem;  // static member variable

        void mf1() { std::cout << "Class2::mf1" << '\n'; }

        struct Nested { Nested() { std::cout << "Nested()" << '\n'; } };

        template <typename T>
        static void print(const T& v) { std::cout << v << '\n'; }

    public:
        explicit Class2(int v) : d_value(v) { }

        void mf2();
        void mf3();
        void mf4();
        void mf5();
    };

    int Class2::s_mem = 0;
}
```

Namespace **ns1** contains three global entities: function **f1**, class **Class1**, and variable **g0**. Namespace **ns2** contains global variables **g1** and **g2**, function **f1**, and classes **Class1** and

C++11

Lambdas

`Class2`. Within `Class2`, we have non-static member variable `d_value`, non-static member function `mf1`, static member function template `print`, and public member functions `mf2` through `mf5`.

With these declarations, we first demonstrate the use of entities that are not variables and are accessible within the scope of a **lambda body**:

```
void ns2::Class2::mf2()
{
    using LocalType = const char*;

    auto c1 = []{
        // Access non-variables in scope
        f1();           // find global ns2::f1 by unqualified name lookup.
        Class1<int> x1; // construct ns2::Class1<int> object.
        Nested      x2; // construct ns2::Class2::Nested object.
        print("print"); // call static member function ns2::Class2::print.
        LocalType   x3; // declare object of local type.
        ns1::f1();     // find global ns1::f1 func by qualified name lookup
        ns1::Class1 x4; // find global ns1::Class1 type by qualified lookup
    };
}
```

We can see that, within the **lambda body**, non-variables can be accessed normally, using either unqualified name lookup or, if needed, qualified name lookup. Unqualified name lookup will find global entities within the namespace; types and static functions within the class; and types declared within the enclosing function scope. Qualified name lookup will find entities in other namespaces.

Variables with static storage duration can also be accessed directly, without being captured:

```
void ns2::Class2::mf3()
{
    static int s1 = 3;
    auto c1 = []{
        // Access variables with static storage duration.
        print(g1); // print global constant ns2::g1
        print(g2); // print global variable ns2::g2
        print(ns1::g0); // print global variable ns1::g0
        print(s_mem); // print static member variable s_mem
        print(s1); // print static local variable s1
    };
}
```

Here we see global constants, global variables, static member variables, and local static variables being used from the local scope.

Next, we look at uses of variables with *automatic* storage duration from the **lambda expression**’s surrounding block scope:

```
void ns2::Class2::mf4()
{
```

```
int      a = 4;
int      b = 5;
int const k = 6;

auto c1 = [](double b) {
    // Access local variables within the reaching scope.
    print(a);    // Error, a is ODR-used but not captured.
    print(b);    // OK, b argument hides b defined in mf1.
    int kb = k;  // OK, const variable is not ODR-used.
    print(k);    // Error, k is ODR-used but not captured.
};
}
```

The attempt to print **a** will fail because **a** is a non-static local variable within the surrounding scope but is not captured. Printing **b** is not a problem because the **b** parameter in the **lambda declarator** is local to the **lambda body**; it hides the **b** variable in the surrounding block scope. A **const** variable of fundamental type is not **ODR-used** unless its *identity* (i.e., address) is needed. Hence, **k** can be used even though it is not captured. Conversely, the call to **print(k)** is an **ODR-use** of **k** because **print** takes its argument by *reference*, which requires taking the *address* of **k**, which, in turn, makes it an **ODR-use**. Since **k** is a local variable with automatic storage duration that was not captured, **print(k)** is ill formed.

Finally, we look at access to (non-static) members of the surrounding class:

```
void ns2::Class2::mf5()
{
    auto c1 = []{
        // Access non-static members
        print(d_value); // Error, this is ODR-used but not captured.
        mf1();          // Error, " " " " " " "
    };
    auto c2 = [=] {
        print(d_value); // OK, this is captured; print this->d_value.
        mf1();          // OK, " " " " ; call this->mf1().
    };
}
```

The above shows that member variables and functions can be accessed only if **this** is (implicitly or explicitly) captured by the **lambda expression**, as is the case for **c2** but not for **c1**.

Use Cases

use-cases-lambda

Interface adaptation, partial application, and currying

ication, -and-currying

Lambda expressions can be used to adapt the set of arguments provided by an algorithm to the parameters expected by another facility:

```
#include <algorithm> // std::count_if
#include <string>     // std::string
#include <vector>     // std::vector
```


C++11

Lambdas

```
extern "C" int f1(const char* s, std::size_t n);

void f2(const std::vector<std::string>& vec)
{
    std::size_t n = std::count_if(vec.begin(), vec.end(),
        [](const std::string& s){ return 0 != f1(s.data(), s.size()); });
    // ...
}
```

Here we have a function, `f1`, that takes a C string and length and computes some predicate, returning 0 for false and nonzero for true. We want to use this predicate with `std::count_if` to count how many strings in a specified vector match this predicate. The **lambda expression** in `f2` adapts `f1` to the needs of `std::count_if` by converting a `std::string` argument into **`const char*`** and `std::size_t` arguments and converting the **`int`** return value to **`bool`**.

A particularly common kind of interface adaptation is **partial application**, whereby we reduce the *parameter count* of a function by holding one or more of its arguments constant for the duration of the algorithm:

```
#include <algorithm> // std::all_of

template <typename InputIter, typename T>
bool all_greater_than(InputIter first, InputIter last, const T& v)
    // returns true if all the values in the specified range [first, last)
    // are greater than the specified v, and false otherwise
{
    return std::all_of(first, last, [&](const T& i) { return i > v; });
}
```

In the example above, the greater-than operator (`>`) takes two operands, but the `std::all_of` algorithm expects a functor taking a single argument. The **lambda expression** passes its single argument as the first operand to **`operator>`** and *binds* the other operand to the captured `v` value, thus solving the interface mismatch.

Finally, let’s touch on **currying**, a transformation borrowed from lambda calculus and functional programming languages. Currying is a flexible way to get results similar to **partial application** by transforming, e.g., a function taking two parameters into one taking just the first parameter and returning a function taking just the second parameter. To apply this technique, we define a **lambda expression** whose call operator returns another **lambda expression**, i.e., a **closure** that returns another **closure**:

```
all_of

template <typename InputIter, typename T>
bool all_greater_than2(InputIter first, InputIter last, const T& v)
    // returns true if all the values in the specified range [first, last)
    // are greater than the specified v, and false otherwise
{
    auto isGreaterThan = [](const T& v){
        return [&](const T& i){ return i > v; };
    };
    return std::all_of(first, last, isGreaterThan(v));
}
```

}

The example above is another way to express the previous example. The call operator for `isGreaterThan` takes a single argument, `v`, and returns another single-argument **closure object** that can be used to compare `i` to `v`. Thus, `isGreaterThan(v)(i)` is equivalent to `i > v`.

Emulating local functions

ating-local-functions

Local functions in languages other than C++ allow functions to be defined within other functions. They are useful when the outer function needs to repeat a set of steps two or more times but where the repeated steps are meaningless outside of the immediate context and/or require access to the outer function’s local variables. Using a **lambda expression** to produce a re-usable **closure** provides this functionality in C++:

```
class Token { /* ... */ };

bool parseToken(const char*& cursor, Token& result)
    // Parse the token at cursor up to the next space or end-of-string,
    // setting result to the resulting token value. Advance cursor to the
    // space or to the null terminator and return true on success. Reset
    // cursor to its original value, set result to an empty token, and
    // return false on failure.
{
    const char* const initCursor = cursor;
    auto error = [&]
    {
        cursor = initCursor;
        result = Token{};
        return false;
    };
    // ...
    if (*cursor++ != '.')
    {
        return error();
    }
    // ...
}
```

The **error closure object** acts as a local function that performs all of the necessary error processing and returns **false**. Using this object, every error branch can be reduced to a single statement, **return error()**. Without **lambda expressions**, the programmer would likely resort to defining a custom class to store the parameters, using a **goto**, or, worse, cutting-and-pasting the three statements shown within the **lambda body**.

Emulate user-defined control constructs

ed-control-constructs

Using a **lambda expression**, an algorithm can look almost like a new control construct in the language:

```
#include <mutex> // std::mutex
```

C++11

Lambdas

```
#include <vector> // std::vector

template <typename RandomIter, typename F>
void parallel_foreach(RandomIter first, RandomIter last, const F& op)
    // For each element, e, in [first, last), create a copy opx of op,
    // and invoke opx(e). Any number of invocations of opx(e) may occur
    // concurrently, each using a separate copy of op.
{ /*...*/ }

void processData(std::vector<double>& data)
{
    double      beta      = 0.0;
    double const init = 7.45e-4;
    std::mutex m;

    parallel_foreach(data.begin(), data.end(), [&, init](double e) mutable
    {
        if (e < 1.0)
        {
            // ...
        }
        else
        {
            // ...
        }
    });
}
```

The `parallel_foreach` algorithm is intended to act like a **for** loop except that all of the elements in the input range may potentially be processed in parallel. By inserting the “body” of this “parallel for loop” directly into the call to `parallel_foreach`, the resulting loop looks and feels a lot like a built-in control construct. Note that the **capture default** is **capture by reference** and will result in all of the iterations sharing the outer function’s call frame, including the mutex variable, `m`, used to prevent race conditions. This **capture default** should be used with care in parallel computations, which often use **capture by copy** to deliberately *avoid* sharing. If an asynchronous computation might outlive its caller, then using **capture by copy** is a must for avoiding dangling references; see *Potential Pitfalls* on page 38.

Variables and control constructs in expressions

Constructs-in-expressions

In situations where a single expression is required — e.g., member-initializers, initializers for **const** variables, and so on — an *immediately evaluated lambda expression* allows that expression to include local variables and control constructs such as loops:

```
#include <climits> // SHRT_MAX

bool isPrime(long i);
    // Return true if i is a prime number.
```

```
const short largestShortPrime = []{
    for (short v = SHRT_MAX; ; v -= 2) {
        if (isPrime(v)) return v;
    }
}();
```

The value of `largestShortPrime` must be set at initialization time because it is a **const** variable with static duration. The loop inside of the **lambda expression** computes the desired value, using local variable, `v`, and a **for** loop. Note that the call operator for the resulting **closure object** is immediately invoked via the `()` argument list at the end of the **lambda expression**; the **closure object** is never stored in a named variable and goes out of scope as soon as the full expression is completely evaluated. This computation would formerly have been possible only by creating a single-use *named* function.

Use with `std::function`

se-with-std::function

As convenient as a **lambda expression** is for passing a functor to an algorithm *template*, the fact that each **closure type** is unnamed and distinct makes it difficult to use them outside of a generic context. The C++11 Standard Library class template, `std::function`, bridges this gap by providing a polymorphic invocable type that can be constructed from any type with a compatible invocation prototype, including but not limited to **closure types**.

A simple interpreter for a postfix input language stores a sequence of instructions in a `std::vector`. Each instruction can be of a different type, but they all accept the current stack pointer as an argument and return the new stack pointer as a result. Each instruction is typically a small operation, ideally suited for being expressed as **lambda expressions**:

```
#include <cstdlib>      // std::strtol
#include <functional>   // std::function
#include <string>       // std::string
#include <vector>       // std::vector

using Instruction = std::function<long*(long*& sp)>;

std::vector<Instruction> instructionStream;

std::string nextToken();           // read the next token
char tokenOp(const std::string& token); // operator for token

void readInstructions()
{
    std::string token;
    Instruction nextInstr;
    while (!(token = nextToken()).empty())
    {
        switch (tokenOp(token))
        {
            case 'i':
            {
```

C++11

Lambdas

```

        // Integer literal
        long v = std::strtol(token.c_str(), nullptr, 10);
        nextInstr = [v](long* sp){ *sp++ = v; return sp; };
        break;
    }
    case '+':
    {
        // + operation
        nextInstr = [](long*& sp){
            long v1 = *--sp;
            long v2 = *--sp;
            *sp++ = v1 + v2;
            return sp;
        };
        break;
    }
    // ... more cases
}

instructionStream.push_back(nextInstr);
}
}

```

The `Instruction` type alias is a `std::function` that can hold, through a process called **type erasure**, any invocable object that takes a `long*` argument and returns a `long*` result. The `readInstructions` function reads successive string tokens and switches on the operation represented by the token. If the operation is `i`, then the token is an integer literal. The string token is converted into a `long` value, `v`, which is captured in a **lambda expression**. The resulting **closure** object is stored in the `nextInstr` variable; when called, it will push `v` onto the stack. Note that the `nextInstr` variable outlives the primary `v` variable, but, because `v` was **captured by copy**, the **captured variable**’s lifetime is the same as the **closure object**’s. If the next operation is `+`, `nextInstr` is set to the **closure object** of an entirely different **lambda expression**, one that captures nothing and whose call operator pops two values from the stack and pushes their sum back onto the stack.

After the **switch** statement, the current value of `nextInstr` is appended to the instruction stream. Note that, although each **closure type** is different, they all can be stored in an `Instruction` object because the prototype for their call operator matches the prototype specified in the instantiation of `std::function`. The `nextInstr` variable can be created empty, assigned from the value of a **lambda expression**, and then later reassigned from the value of a different **lambda expression**. This flexibility makes `std::function` and **lambda expressions** a potent combination.

One specific use of `std::function` worth noting is to return a **lambda expression** from a non-template function:

```

std::function
std::function<int(int)> add_n(int n)
{
    return [n](int i) { return n + i; };
}

```

```

}

int result = add_n(3)(5); // result is 8.

```

The return value of `add_n` is a **closure object** wrapped in a `std::function` object. Note that `add_n` is not a template and that it is not called in a template or **auto** context. This example illustrates a runtime-polymorphic way to achieve **currying**; see the earlier example in *Interface adaptation, partial application, and currying* on page 30.

Event-driven callbacks

Event-driven systems tend to have interfaces that are littered with callbacks:

```

#include <memory> // std::unique_ptr

class DialogBox { /* ... */ };

template <typename Button1Func, typename Button2Func>
std::unique_ptr<DialogBox> twoButtonDialog(const char* prompt,
                                          const char* button1text,
                                          Button1Func button1callback,
                                          const char* button2text,
                                          Button2Func button2callback)
{
    // ...
}

```

The `twoButtonDialog` factory function takes three strings and two callbacks and returns a pointer to a dialog box having two buttons. The dialog-box logic invokes one of the two callbacks, depending on which of the two buttons is pressed. These callbacks are often quite small pieces of code that can best be expressed directly in the program logic using **lambda expressions**:

```

void runModalDialogBox(DialogBox& db);

void launchShuttle(/* ... */)
{
    bool doLaunch = false;

    std::unique_ptr<DialogBox> confirm =
        twoButtonDialog("Are you sure you want to launch the shuttle?",
                        "Yes", [&]{ doLaunch = true; },
                        "No", []{});

    runModalDialogBox(*confirm);

    if (doLaunch)
    {
        // ... launch the shuttle!
    }
}

```

Here, the user is being prompted as to whether or not to launch a missile. Since the dialog box is processed entirely within the `launchShuttle` function, it is convenient to express two callbacks in-place, within the function, using **lambda expressions**. The first **lambda expression** — passed as the callback for when the user clicks “Yes” — captures the `doLaunch` flag by reference and simply sets it to **true**. The second **lambda expression** — passed as the callback for when the user clicks “No” — does nothing, leaving the `doLaunch` flag having its original **false** value. The simplicity of these callbacks come from fact that they are effectively extensions of the surrounding block and, hence, have access (via the **lambda capture**) to block-scoped variables such as `doLaunch`.

recursion Recursion

A **lambda expression** cannot refer to itself, so creating one that is recursive involves using one of a number of different possible workarounds. If the **lambda capture** is empty, recursion can be accomplished fairly simply by converting the **lambda expression** into a plain function pointer stored in a **static** variable:

```
void f1()
{
    static int (*const fact)(int) = [](int i)
    {
        return i < 2 ? 1 : i * fact(i-1);
    };

    int result = fact(4); // computes 24
}
```

In the above example, `fact(n)` returns the factorial of `n`, computed using a recursive algorithm. The variable, `fact`, becomes visible before its initializer is compiled, allowing it to be called from within the **lambda expression**. To enable the conversion to function pointer, the **lambda capture** must be empty; hence, `fact` must be static so that it can be accessed without capturing it.

If a recursive **lambda expression** is desired with a nonempty **lambda capture**, then the entire recursion can be enclosed in an outer **lambda expression**:

```
void f2(int n)
{
    auto permsN = [n](int m) -> int
    {
        static int (*const imp)(int, int) = [](int x, int m) {
            return m <= x ? m : m * imp(x, m - 1);
        };
        return imp(m - n + 1, m);
    };

    int a = permsN(5); // permutations of 5 items, n at a time
    int b = permsN(4); // permutations of 4 items, n at a time
}
```

In this example, `permsN(m)`, returns the number of permutations of `m` items taken `n` at a time, where `n` is captured by the **closure object**. The implementation of `permsN` defines a nested `imp` function pointer that uses the same technique as `fact`, above, to achieve recursion. Since `imp` must have an empty **lambda capture**, everything it needs is passed in as arguments by the `permsN` enclosing **lambda expression**. Note that the `imp` pointer and the **lambda expression** from which it is initialized do not need to be scoped inside of the `permN` **lambda expression**; whether such nesting is desirable is a matter of taste.

In C++14, additional approaches to recursion (e.g., the “Y Combinator” borrowed from lambda calculus⁷) are possible due to using **generic lambdas**; see Section 2.2:“??” on page ??.

Potential Pitfalls

Dangling references

Closure objects can capture references to local variables and copies of the **this** pointer. If a copy of the **closure object** outlives the stack frame in which it was created, these references can refer to objects that have been destroyed. The two ways in which a **closure object** can outlive its creation context are if (1) it is returned from the function or (2) it is stored in a data structure for later invocation:

```
#include <functional> // std::function
#include <vector>      // std::vector

class Class1
{
    int d_mem;

    static std::vector<std::function<double(void*)> > s_workqueue;

    std::function<void(int)> mf1()
    {
        int local;
        return [&](int i) -> void { d_mem = local = i; }; // Bug, dangling refs
    }

    void mf2()
    {
        double local = 1.0;
        s_workqueue.push_back([&, this](void* p) -> double {
            return p ? local : double(d_mem);
        }); // Bug, dangling refs
    }
};
```

The example above uses `std::function` to hold **closure objects**, as described in *Use with std::function* on page 34. In member function `mf1`, the **lambda body** modifies both the local variable and the member variable currently in scope. However, as soon as the function returns, the local variable goes out of scope and the **closure** contains a dangling reference.

⁷?

Moreover, the object on which it is invoked can also go out of scope while the **closure object** continues to exist. Modifying either **this->d_mem** or **local** through the capture is likely to corrupt the stack, leading to a crash, potentially much later in the program.

The member function **mf2**, rather than *return* a **closure** with dangling references, stores it in a data structure, i.e., the **s_workqueue** static **vector**. Once again, **local** and **d_mem** become dangling references and can result in data corruption when the call operator for the stored **closure object** is invoked. It is safest to capture **this** and use **capture by reference** only when the lifetime of the **closure object** is clearly limited to the current function. **Implicitly captured this** is particularly insidious because, even if the **capture default** is **capture by copy**, member variables are not copied and are often referenced without the **this->** prefix, making them hard to spot in the source code.

overuse Overuse

The ability to write functions, especially functions with state, at the point where they are needed and without much of the syntactic overhead that accompanies normal functions and class methods, can potentially lead to a style of code that uses “lambdas everywhere”, losing the abstraction and well-documented interfaces of separate functions. **Lambda expressions** are not intended for large-scale reuse. Sprinkling **lambda expressions** throughout the code can result in poor software-engineering practices such as cut-and-paste programming and the absence of cohesive abstractions.

non-captured-variables Mixing captured and non-captured variables

A **lambda body** can access both automatic-duration local variables that were captured from the enclosing block and static-duration variables that need not and cannot be captured. Variables **captured by copy** are “frozen” at the point of capture and cannot be changed except by the **lambda body** (if mutable), whereas static variables can be changed independent of the **lambda expression**. This difference is often useful but can cause confusion when reasoning about a **lambda expression**:

```
assert
```

```
void f1()
{
    static int a;
    int      b;

    a = 5;
    b = 6;

    auto c1 = [b]{ return a + b; }; // OK, b is **captured by copy**.
    assert(11 == c1());             // OK, a == 5 and b == 6.
    ++b;                            // Increment *primary* b.
    assert(11 == c1());             // OK, captured b did not change.
    ++a;                            // Increment static-duration a.
    assert(12 == c1());             // Bug, a == 6 and captured b == 6
}
```

When the **closure object** for `c1` is created, the captured `b` value is frozen within the **lambda body**. Changing the primary `b` has no effect. However, `a` is not captured (nor is it allowed to be). As a result, there is only *one* `a` variable, and modifying that variable outside of the **lambda body** changes the result of invoking the call operator.

C++14 **capture init** can be used to effectively capture a non-local variable:

```
assert

void f2()
{
    static int a;
    int      b;

    a = 5;
    b = 6;

    auto c1 = [a=a,b]{ return a + b; }; // OK, capture both a and b.
    ++a;                                // Increment static-duration a.
    assert(11 == c1());                 // OK, captured a did not change.
}
```

The **lambda capture**, `[a=a,b]`, creates a new capture variable `a` that is initialized from the static variable `a` at the point that the **lambda expression** is evaluated to produce a **closure object**. The `a` variable within the **lambda body** refers to this captured variable, not to the static one.

Local variables in unevaluated contexts can yield surprises

To use a local variable, `x`, from the surrounding block as part of an unevaluated operand (e.g., **`sizeof(x)`** or **`alignof(x)`**), it is generally not necessary to capture `x` because it is not **ODR-used** within the **lambda body**. Whether or not `x` is captured, most expressions in unevaluated contexts behave as though `x` were *not* captured and the expression were evaluated directly in the enclosing block scope. This is itself surprising because, for example, a captured variable in a non-**mutable lambda expression** is **const**, whereas the primary variable might not be:

```
#include <iostream> // std::cout

short s1(int&)      { return 0; }
int    s1(const int&) { return 0; }

void f1()
{
    int x = 0; // x is a non-const lvalue.
    [x]{
        // captured x in non-mutable lambda is lvalue of type const int
        std::cout << sizeof(s1(x)) << '\n'; // prints sizeof(short)
        auto s1x = s1(x);                  // yields an int
        std::cout << sizeof(s1x) << '\n';   // prints sizeof(int)
    }();
}
```

The first print statement calls `s1(x)` in an unevaluated context, which ignores the captured `x` and returns the size of the result of `s1(int&)`. The next statement actually *evaluates* `s1(x)`, passing the *captured* `x` and calling `s1(const int&)` because the call operator is not decorated with **mutable**.

When using `decltype(x)`, the result is the declared type of the *primary* variable, regardless of whether or not `x` was captured. However, if `x` had been *captured by copy*, `decltype(x)` (with two sets of parentheses) would have yielded the *lvalue* type of the *captured* variable. There is some dispute as to what the correct results should be if `x` is *not* captured, with some compilers yielding the type of the primary variable and others complaining that it was not captured.

```
void f1()
{
    int x = 0; // x is a non-const lvalue.
    auto c1 = [x]{ decltype(x) y = x; }; // y has type const int&.
}
```

Finally, there is an unsettled question as to whether `typeid(x)` is an *ODR-use* of `x` and, therefore, requires that `x` be captured. Some compilers will complain about the following code:

```
#include <typeinfo> // typeid
void f3()
{
    int x = 0;
    auto c1 = []{ return typeid(x); };
    // Error, on some platforms `x was not captured'
}
```

One can avoid this pitfall simply by calling `typeid` outside of the lambda, capturing the result if necessary:

```
#include <typeinfo> // typeid
void f3()
{
    int x = 0;
    const std::type_info& xid = typeid(x); // OK, typeid called outside of lambda
    auto c1 = [&]() -> const std::type_info&
    { return xid; }; // OK, return captured typeid
}
```

Annoyances

annoyances

Debugging

debugging

By definition, lambdas do not have names. Tools such as debuggers and stack-trace examiners typically display the compiler-generated names of the closure types instead of names selected by the programmer to clearly describe the purpose of a function, making it difficult to discern where a problem occurred.

capture-**this*-by-copy

Can’t capture **this* by copy

A **lambda expression** can freeze the value of a surrounding local variable by using **capture by copy**, but no such ability is available directly to copy the object pointed to by **this**. In C++14, this deficiency can be mitigated using **capture init**:

```
class Class1
{
    int d_value;

    void mf1()
    {
        auto c1 = [self=*this]{ return self.d_value; };
    }
};
```

The **lambda capture**, **[self=**this*]** creates a new captured variable, **self**, that contains a copy of ***this**. Unfortunately, accessing member variable **d_value** requires explicit use of **self.d_value**.

C++11 doesn’t have **capture init**, so it is necessary to create a **self** variable external to the **lambda expression** and capture *that* variable⁸:

```
class Class1
{
    int d_value;

    void mf1()
    {
        Class1& self = *this;
        auto c1 = [self]{ return self.d_value; };
    }
};
```

Deferred-execution-code

Confusing mix of immediate and deferred-execution code

The main selling point of **lambda expressions** — i.e., the ability to define a function object at the point of use — can sometimes be a liability. The code within a **lambda body** is

⁸As of C++17, ***this** can be captured directly with **this** within the **lambda body** pointing to the *copy* rather than the original:

```
class Class2
{
    int d_value;

    void mf1()
    {
        auto c1 = [*this]{ return d_value; };
        // C++17: return d_value from copy of *this
    }
};
```

C++11

Lambdas

typically not executed immediately but is deferred until some other piece of code, e.g., an algorithm, invokes it as a callback. The code that is immediately executed and the code whose invocation is deferred are visually intermixed in a way that could confuse a future maintainer. For example, let’s look at a simplified excerpt from an earlier use case, *Use Cases — Use with std::function* on page 34.

```
#include <string>      // std::string
#include <functional>  // std::function

void readInstructions()
{
    std::string          token;
    std::function<long*(long*& sp)> nextInstr;

    while ( /* ... */ (!token.empty()))
    {
        switch (token[0])
        {
            // ... more cases
            case '+':
            {
                // + operation
                nextInstr = [](long*& sp){
                    long v1 = *--sp;
                    long v2 = *--sp;
                    *sp++ = v1 + v2;
                    return sp;
                };
                break;
            }
            // ... more cases
        }
        // ...
    }
}
```

A casual reading might lead to the assumption that operations such as `*--sp` are taking place within **case '+'**, when the truth is that these operations are encapsulated in a **lambda expression** and are not executed until the **closure object** is called (via `nextInstr`) in a relatively distant part of the code.

Trailing punctuation

trailing-punctuation

The body of a **lambda expression** is a *compound statement*. When compound statements appear elsewhere in the C++ grammar, e.g., as the body of a function or loop, they are not followed by punctuation. A **lambda expression**, conversely, is invariably followed by some sort of punctuation, usually a semicolon or parenthesis but sometimes a comma or binary operator. This difference between a **lambda body** and other compound statements makes this punctuation easy to forget:

```
auto c1 = []{ /* ... */ }; // <-- Don't forget the semicolon at the end.
```

The extra punctuation can also be unattractive when emulating a control construct using **lambda expressions** as in the `parallel_foreach` example in *Use Cases — Emulate user-defined control constructs* on page 32:

```
std::vector

void f(const std::vector<int>& data)
{
    // ...
    for (int e : data)
    {
        // ...           for loop body
    } // <-- no punctuation after the closing brace

    parallel_foreach(data.begin(), data.end(), [&](int e)
    {
        // ...           parallel loop body
    }); // <-- Don't forget the closing parenthesis and semicolon.
    // ...
}
```

In the above code snippet, the programmer would like the `parallel_foreach` algorithm to look as much like the built-in **for** loop as possible. However, the built-in **for** loop doesn’t end with a closing parenthesis and a semicolon, whereas the `parallel_foreach` does, so the illusion of a language extension is incomplete.

see-also

See Also

- “??” (§1.1, p. ??) ♦ illustrates a form of type inference often used in conjunction with (or in place of) trailing return types.
- “??” (§3.2, p. ??) ♦ shows a form of type inference that shares syntactical similarities with trailing return types, leading to potential pitfalls when migrating from C++11 to C++14.

further-reading

Further Reading

TODO

C++14

Lambdas

sec-conditional-cpp14

Chapter 3

Unsafe Features

ch-unsafe
sec-unsafe-cpp11

Intro text should be here.

Chapter 3 Unsafe Features

sec-unsafe-cpp14