Chapter 1

Safe Features

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

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

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)
```

```
    // 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));
```

```
}
```

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.

f-a-lambda-expression

Parts of 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."Lambda Captures" on page 55) 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([]{/* ... */});  // 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
```

[1]The 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 **?**.

[2]C++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."Lambda Captures" on page 55), 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::cot
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
{
```

[3]C++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]
    {
```

```
        ++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:

```
    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
    {
```

[4]In 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()
            {
```

```
                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 { /* ... */ }
           \_________ ________/ \__ __/   \________ ________/   \_____ ___/
                     V               V                V                  V
            parameter list   mutability    exception spec     return type
           \_____________________________ _____________________________/
                                         V
                                  lambda declarator
```

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
```

---

[5]A 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 **const**ness: 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 **const**ness 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 non**`const`** 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]

[6]The 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

`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-lambda

Use Cases

ication,-and-currying

Interface adaptation, partial application, 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
```

```
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 [&v](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
```

```
#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

nstructs-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.

se-with-std::function

Use 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':
            {
```

```
                // 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.

vent-driven-callbacks

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 needed 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 calculus[7]) are possible due to using generic lambdas; see Section 2.2."??" on page ??.

ntial-pitfalls-lambda

Potential Pitfalls

dangling-references

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

s-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 = []() -> const std::type_info& { 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 typeinfo
}
```

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* variable[8]:

```
class Class1
{
    int d_value;

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

ferred-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

[8]As 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
    }
};
```

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

sec-conditional-cpp14

Relaxed Restrictions on `constexpr` Functions

onstexpr-restrictions

C++14 lifts restrictions regarding use of many language features in the body of a `constexpr` function (see "**??**" on page **??**).

Description

description

The cautious introduction (in C++11) of `constexpr` functions — i.e., functions eligible for compile-time evaluation — was accompanied by a set of strict rules that, despite making life easier for compiler implementers, severely narrowed the breadth of valid use cases for the feature. In C++11, `constexpr` function bodies were restricted to essentially a single `return` statement and were not permitted to have any modifiable local state (variables) or **imperative** language constructs (e.g., assignment), thereby greatly reducing their usefulness:

```
constexpr int fact11(int x)
{
    static_assert(x >= 0, "");
        // Error, x is not a constant expression.

    static_assert(sizeof(x) >= 4, "");  // OK in C++11/14

    return x < 2 ? 1 : x * fact11(x - 1);  // OK in C++11/14
}
```

Notice that recursive calls were supported, often leading to convoluted implementations of algorithms (compared to an **imperative** counterpart); see *Use Cases: Nonrecursive* `constexpr` *algorithms* on page 48.

The C++11 `static_assert` feature (see "**??**" on page **??**) was always permitted in a C++11 `constexpr` function body. However, because the input variable `x` in `fact11` (in the code snippet above) is inherently not a compile-time constant expression, it can never appear as part of a `static_assert` predicate. Note that a `constexpr` function returning `void` was also not permitted:

```
constexpr void no_op() { }  // Error in C++11; OK in C++14
```

Experience gained from the release and subsequent real-world use of C++11 emboldened the standard committee to lift most of these (now seemingly arbitrary) restrictions for C++14, allowing use of (nearly) *all* language constructs in the body of a `constexpr` function. In C++14, familiar non-expression-based control-flow constructs, such as `if` statements and `while` loops, are also available, as are modifiable local variables and assignment operations:

```
constexpr int fact14(int x)
{
    if (x <= 2)        // Error in C++11; OK in C++14
    {
        return 1;
    }
```

```
    int temp = x - 1;  // Error in C++11; OK in C++14
    return x * fact14(temp);
}
```

Some useful features remain disallowed in C++14; most notably, any form of dynamic allocation is not permitted, thereby preventing the use of common standard container types, such as `std::string` and `std::vector`[1]:

1. `asm` declarations
2. `goto` statements
3. Statements with labels other than `case` and `default`
4. `try` blocks
5. Definitions of variables
   (a) of other than a **literal type** (i.e., fully processable at compile time)
   (b) decorated with either `static` or `thread_local`
   (c) left uninitialized

The restrictions on what can appear in the body of a `constexpr` that remain in C++14 are reiterated here in codified form[2]:

```
template <typename T>
constexpr void f()
{
try {                   // Error, try outside body isn't allowed (until C++20).
    std::ifstream is;   // Error, objects of *non-literal* types aren't allowed.
    int x;              // Error, uninitialized vars. disallowed (until C++20)
    static int y = 0;   // Error, static variables are disallowed.
    thread_local T t;   // Error, thread_local variables are disallowed.
    try{}catch(...){}   // Error, try/catch disallowed (until C++20)
    if (x) goto here;   // Error, goto statements are disallowed.
    []{};               // Error, lambda expressions are disallowed (until C++17).
here: ;                 // Error, labels (except case/default) aren't allowed.
    asm("mov %r0");     // Error, asm directives are disallowed.
} catch(...) { }        // Error, try outside body disallowed (until C++20)
}
```

[1]In C++20, even more restrictions were lifted, allowing, for example, some limited forms of dynamic allocation, `try` blocks, and uninitialized variables.

[2]Note that the degree to which these remaining forbidden features are reported varies substantially from one popular compiler to the next.

ases-relaxedconstexpr

Use Cases

-constexpr-algorithms

Nonrecursive `constexpr` algorithms

The C++11 restrictions on the use of `constexpr` functions often forced programmers to implement algorithms (that would otherwise be implemented iteratively) in a recursive manner. Consider, as a familiar example, a naive[3] C++11-compliant `constexpr` implementation of a function, `fib11`, returning the *n*th Fibonacci number[4]:

```
constexpr long long fib11(long long x)
{
    return
        x == 0 ? 0
               : (x == 1 || x == 2) ? 1
                                    : fib11(x - 1) + fib11(x - 2);
}
```

The implementation of the `fib11` function (above) has various undesirable properties.

1. *Reading difficulty* — Because it must be implemented using a single `return` statement, branching requires a chain of *ternary operators*, leading to a single long expression that might impede human comprehension.

2. *Inefficiency and lack of scaling* — The explosion of recursive calls is taxing on compilers: (1) the time to compile is markedly slower for the *recursive* (C++11) algorithm than it would be for its *iterative* (C++14) counterpart, even for modest inputs,[5] and (2) the compiler might simply refuse to complete the compile-time calculation if it exceeds some internal (platform-dependent) *threshold* number of operations.[6]

---

[3]For a more efficient (yet less intuitive) C++11 algorithm, see *Appendix: Optimized C++11 Example Algorithms, Recursive Fibonacci* on page 53.

[4]We used `long long` (instead of `long`) here to ensure a unique C++ type having at least 8 bytes on all conforming platforms for simplicity of exposition (avoiding an internal copy). We deliberately chose *not* to make the value returned `unsigned` because the extra bit does not justify changing the **algebra** (from `signed` to `unsigned`). For more discussion on these specific topics, see "**??**" on page **??**.

[5]As an example, Clang 10.0.0, running on an x86-64 machine, required more than 80 times longer to evaluate `fib(27)` implemented using the *recursive* (C++11) algorithm than to evaluate the same functionality implemented using the *iterative* (C++14) algorithm.

[6]The same Clang 10.0.0 compiler discussed in the previous footnote failed to compile `fib11(28)`:

```
error: static_assert expression is not an integral constant expression
    static_assert(fib11(28) == 317811, "");
                 ^~~~~~~~~~~~~~~~~~

note: constexpr evaluation hit maximum step limit; possible infinite loop?
```

GCC 10.x fails at `fib(36)`, with a similar diagnostic:

```
error: 'constexpr' evaluation operation count exceeds limit of 33554432
       (use '-fconstexpr-ops-limit=' to increase the limit)
```

Clang 10.x fails to compile any attempt at constant evaluating `fib(28)`, with the following diagnostic message:

```
note: constexpr evaluation hit maximum step limit; possible infinite loop?
```

3. *Redundancy* — Even if the recursive implementation were suitable for small input values during compile-time evaluation, it would be unlikely to be suitable for any runtime evaluation, thereby requiring programmers to provide and maintain *two* separate versions of the same algorithm: a compile-time *recursive* one and a runtime *iterative* one.

In contrast, an *imperative* implementation of a `constexpr` function implementing a function returning the *n*th Fibonacci number in C++14, `fib14`, does not suffer from any of the three issues discussed above:

```
constexpr long long fib14(long long x)
{
    if (x == 0) { return 0; }

    long long a = 0;
    long long b = 1;

    for (long long i = 2; i <= x; ++i)
    {
        long long temp = a + b;
        a = b;
        b = temp;
    }

    return b;
}
```

As one would expect, the compile time required to evaluate the iterative implementation (above) is manageable[7]; of course, far more computationally efficient (e.g., closed form[8]) solutions to this classic exercise are available.

aprogramming-algorithms

Optimized metaprogramming algorithms

C++14's relaxed `constexpr` restrictions enable the use of modifiable local variables and **imperative** language constructs for metaprogramming tasks that were historically often implemented by using (Byzantine) recursive template instantiation (notorious for their voracious consumption of compilation time).

Consider, as the simplest of examples, the task of counting the number of occurrences of a given type inside a **type list** represented here as an empty variadic template (see "**??**" on page **??**) that can be instantiated using a variable-length sequence of arbitrary C++ types[9]:

[7]Both GCC 10.x and Clang 10.x evaluated `fib14(46)` 1836311903 correctly in under 20ms on a machine running Windows 10 x64 and equipped with a Intel Core i7-9700k CPU.

[8]E.g., see http://mathonline.wikidot.com/a-closed-form-of-the-fibonacci-sequence.

[9]Variadic templates are a C++11 feature having many valuable and practical uses. In this case, the variadic feature enables us to easily describe a template that takes an arbitrary number of C++ type arguments by specifying an ellipsis (`...`) immediately following `typename`. Emulating such functionality in C++98/03 would have required significantly more effort: A typical workaround for this use case would have been to create a template having some fixed maximum number of arguments (e.g., 20), each defaulted to some unused (incomplete) type (e.g., `Nil`):

```
template <typename...> struct TypeList { };
    // empty variadic template instantiable with arbitrary C++ type sequence
```

Explicit instantiations of this variadic template could be used to create objects:

```
TypeList<>                   emptyList;
TypeList<int>                listOfOneInt;
TypeList<int, long, double>  listOfThreeIntLongDouble;
```

A naive C++11-compliant implementation of a **metafunction** `Count`, used to ascertain the (order-agnostic) number of times a given C++ type was used when creating an instance of the `TypeList` template (above), would usually make recursive use of (baroque) **partial class template specialization**[10] to satisfy the single-return-statement requirements[11]:

```
struct Nil;  // arbitrary unused (incomplete) type

template <typename = Nil, typename = Nil, typename = Nil, typename = Nil>
struct TypeList { };
    // emulates the variadic TypeList template struct for up to four
    // type arguments
```

Another theoretically appealing approach is to implement a Lisp-like recursive data structure; the compile-time overhead for such implementations, however, often makes them impractical.

[10]The use of class-template specialization (let alone partial specialization) might be unfamiliar to those not accustomed to writing low-level template metaprograms, but the point of this use case is to obviate such unfamiliar use. As a brief refresher, a general class template is what the client typically sees at the user interface. A specialization is typically an implementation detail consistent with the **contract** specified in the general template but somehow more restrictive. A partial specialization (possible for *class* but not *function* templates) is itself a template but with one or more of the general template parameters resolved. An **explicit** or **full specialization** of a template is one in which *all* of the template parameters have been resolved and, hence, is not itself a template. Note that a **full specialization** is a stronger candidate for a match than a partial specialization, which is a stronger match candidate than a simple template specialization, which, in turn, is a better match than the general template (which, in this example, happens to be an **incomplete type**).

[11]Notice that this `Count` **metafunction** also makes use (in its implementation) of variadic class templates to parse a **type list** of unbounded depth. Had this been a C++03 implementation, we would have been forced to create an approximation (to the simple class-template specialization containing the **parameter pack** `Tail...`) consisting of a bounded number (e.g., 20) of simple (class) template specializations, each one taking an increasing number of template arguments:

std::integral_constantstd::is_same

```
template <typename X, typename Y>
struct Count<X, TypeList<Y>>
    : std::integral_constant<int, std::is_same<X, Y>::value> { };
    // (class) template specialization for one argument

template <typename X, typename Y, typename Z>
struct Count<X, TypeList<Y, Z>>
    : std::integral_constant<int,
        std::is_same<X, Y>::value + std::is_same<X, Z>::value> { };
    // (class) template specialization for two arguments

template <typename X, typename Y, typename Z, typename A>
struct Count<X, TypeList<Y, Z, A>>
    : std::integral_constant<int,
        std::is_same<X, Y>::value + Count<X, TypeList<Z, A>>::value> { };
    // recursive (class) template specialization for three arguments
```

axedconstexpr-countcode

```
#include <type_traits>  // std::integral_constant, std::is_same

template <typename X, typename List> struct Count;
    // general template used to characterize the interface for the Count
    // metafunction
    // Note that this general template is an incomplete type.

template <typename X>
struct Count<X, TypeList<>> : std::integral_constant<int, 0> { };
    // partial (class) template specialization of the general Count template
    // (derived from the integral-constant type representing a compile-time
    // 0), used to represent the base case for the recursion --- i.e., when
    // the supplied TypeList is empty
    // The payload (i.e., the enumerated value member of the base class)
    // representing the number of elements in the list is 0.

template <typename X, typename Head, typename... Tail>
struct Count<X, TypeList<Head, Tail...>>
    : std::integral_constant<int,
        std::is_same<X, Head>::value + Count<X, TypeList<Tail...>>::value> { };
    // simple (class) template specialization of the general count template
    // for when the supplied list is not empty
    // In this case, the second parameter will be partitioned as the first
    // type in the sequence and the (possibly empty) remainder of the
    // TypeList. The compile-time value of the base class will be either the
    // same as or one greater than the value accumulated in the TypeList so
    // far, depending on whether the first element is the same as the one
    // supplied as the first type to Count.

static_assert(Count<int, TypeList<int, char, int, bool>>::value == 2, "");
```

Notice that we made use of a C++11 **parameter pack**, `Tail...` (see "**??**" on page **??**), in the implementation of the simple template specialization to package up and pass along any remaining types.

As should be obvious by now, the C++11 restriction encourages both somewhat rarified metaprogramming-related knowledge and a *recursive* implementation that can be compile-time intensive in practice.[12] By exploiting C++14's relaxed `constexpr` rules, a simpler and typically more compile-time friendly *imperative* solution can be realized:

```
std::integral_constantstd::is_same

template <typename X, typename... Ts>
constexpr int count()
{
```

```
// ...
```

[12]For a more efficient C++11 version of `Count`, see *Appendix: Optimized C++11 Example Algorithms*, `constexpr` *type list* `Count` *algorithm* on page 53.

```
    bool matches[sizeof...(Ts)] = { std::is_same<X, Ts>::value... };
        // Create a corresponding array of bits where 1 indicates sameness.

    int result = 0;
    for (bool m : matches)  // (C++11) range-based for loop
    {
        result += m;        // Add up 1 bits in the array.
    }

    return result;  // Return the accumulated number of matches.
}
```

The implementation above — though more efficient and comprehensible — will require some initial learning for those unfamiliar with modern C++ variadics. The general idea here is to use **pack expansion** in a nonrecursive manner[13] to initialize the `matches` array with a sequence of zeros and ones (representing, respectively, mismatch and matches between `X` and a type in the `Ts...` pack) and then iterate over the array to accumulate the number of ones as the final `result`. This `constexpr`-based solution is both easier to understand and typically faster to compile.[14]

potential-pitfalls

Potential Pitfalls

None so far

annoyances

Annoyances

None so far

see-also

See Also

- “??” — Conditionally safe C++11 feature that first introduced compile-time evaluations of functions.

---

[13]**Pack expansion** is a language construct that expands a **variadic pack** during compilation, generating code for each element of the pack. This construct, along with a **parameter pack** itself, is a fundamental building block of variadic templates, introduced in C++11. As a minimal example, consider the variadic function template, `e`:

```
template <int... Is> void e() { f(Is...); }
```

`e` is a function template that can be instantiated with an arbitrary number of compile-time-constant integers. The `int... Is` syntax declares a **variadic pack** of compile-time-constant integers. The `Is...` syntax (used to invoke `f`) is a basic form of pack expansion that will resolve to all the integers contained in the `Is` pack, separated by commas. For instance, invoking `e<0, 1, 2, 3>()` results in the subsequent invocation of `f(0, 1, 2, 3)`. Note that — as seen in the `count` example (which starts on page 51) — any arbitrary expression containing a variadic pack can be expanded:

```
template <int... Is> void g() { h((Is > 0)...); }
```

The `(Is > 0)...` expansion (above) will resolve to `N` comma-separated Boolean values, where `N` is the number of elements contained in the `Is` **variadic pack**. As an example of this expansion, invoking `g<5, -3, 9>()` results in the subsequent invocation of `h(true, false, true)`.

[14]For a type list containing 1024 types, the imperative (C++14) solution compiles about twice as fast on GCC 10.x and roughly 2.6 times faster on Clang 10.x.

- "??" — Conditionally safe C++11 feature that first introduced variables usable as constant expressions.
- "??" — Conditionally safe C++11 feature allowing templates to accept an arbitrary number of parameters.

Further Reading

further-reading

None so far

Appendix: Optimized C++11 Example Algorithms

++11-example-algorithms

Recursive Fibonacci

recursive-fibonacci

Even with the restrictions imposed by C++11, we can write a more efficient recursive algorithm to calculate the nth Fibonacci number:

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

constexpr std::pair<long long, long long> fib11NextFibs(
    const std::pair<long long, long long> prev,  // last two calculations
    int count)                                   // remaining steps
{
    return (count == 0) ? prev : fib11NextFibs(
        std::pair<long long, long long>(prev.second,
                                        prev.first + prev.second),
        count - 1);
}

constexpr long long fib11Optimized(long long n)
{
    return fib11NextFibs(
        std::pair<long long, long long>(0, 1), // first two numbers
        n                                      // number of steps
    ).second;
}
```

`constexpr` type list `Count` algorithm

ypelist-count-algorithm

As with the `fib11Optimized` example, providing a more efficient version of the `Count` algorithm in C++11 is also possible, by accumulating the final result through recursive `constexpr` function invocations:

```
#include <type_traits>  // std::is_same

template <typename>
constexpr int count11Optimized() { return 0; }
    // Base case: always return 0.

template <typename X, typename Head, typename... Tail>
constexpr int count11Optimized()
```

```
    // Recursive case: compare the desired type (X) and the first type in
    // the list (Head) for equality, turn the result of the comparison
    // into either 1 (equal) or 0 (not equal), and recurse with the rest
    // of the type list (Tail...).
{
    return (std::is_same<X, Head>::value ? 1 : 0)
        + count11Optimized<X, Tail...>();
}
```

This algorithm can be optimized even further in C++11 by using a technique similar to the one shown for the iterative C++14 implementation. By leveraging a `std::array` as compile-time storage for bits where `1` indicates equality between types, we can compute the final result with a fixed number of template instantiations:

```
#include <array>        // std::array
#include <type_traits>  // std::is_same

template <int N>
constexpr int count11VeryOptimizedImpl(
    const std::array<bool, N>& bits,  // storage for "type sameness" bits
    int i)                            // current array index
{
    return i < N
        ? bits[i] + count11VeryOptimizedImpl<N>(bits, i + 1)
            // Recursively read every element from the bits array and
            // accumulate into a final result.
        : 0;
}

template <typename X, typename... Ts>
constexpr int count11VeryOptimized()
{
    return count11VeryOptimizedImpl<sizeof...(Ts)>(
        std::array<bool, sizeof...(Ts)>{ std::is_same<X, Ts>::value... },
            // Leverage pack expansion to avoid recursive instantiations.
        0);
}
```

Note that, despite being recursive, `count11VeryOptimizedImpl` will be instantiated only once with `N` equal to the number of elements in the `Ts...` pack.

Lambda-Capture Expressions

lambda-capture-expressions

Lambda-capture expressions enable **synthetization** (spontaneous implicit creation) of arbitrary data members within **closures** generated by lambda expressions (see "Lambdas" on page 4).

Description

description

In C++11, lambda expressions can capture variables in the surrounding scope either *by value* or *by reference*[1]:

```
void test()
{
    int i = 0;
    auto f0 = [i]{ };   // Create a copy of i in the closure named f0.
    auto f1 = [&i]{ };  // Store a reference to i in the closure named f1.
}
```

Although one could specify *which* and *how* existing variables were captured, the programmer had no control over the creation of new variables within a **closure**. C++14 extends the **lambda-introducer** syntax to support implicit creation of arbitrary data members inside a **closure** via either **copy initialization** or **list initialization**:

```
auto f2 = [i = 10]{ /* body of closure */ };
    // Synthesize an int data member, i, initialized with 10 in the closure.

auto f3 = [c{'a'}]{ /* body of closure */ };
    // Synthesize a char data member, c, initialized with 'a' in the closure.
```

Note that the identifiers `i` and `c` above do not refer to any existing variable; they are specified by the programmer creating the closure. For example, the **closure** type assigned (i.e., bound) to `f2` (above) is similar in functionality to an **invocable** `struct` containing an `int` data member:

```
// pseudocode
struct f2LikeInvocableStruct
{
    int i = 10;  // The type int is deduced from the initialization expression.
    auto operator()() const { /* closure body */ }  // The struct is invocable.
};
```

The type of the data member is deduced from the initialization expression provided as part of the capture in the same vein as `auto` (see "**??**" on page **??**) type deduction; hence, it's not possible to synthesize an uninitialized **closure** data member:

```
auto f4 = [u]{ };    // Error, u initializer is missing for lambda capture.
auto f5 = [v{}]{ };  // Error, v's type cannot be deduced.
```

[1] We use the familiar (C++11) feature `auto` (see "**??**" on page **??**) to deduce a closure's type since there is no way to name such a type explicitly.

It is possible, however, to use variables outside the scope of the lambda as part of a lambda-capture expression (even capturing them *by reference* by prepending the `&` token to the name of the synthesized data member):

```
int i = 0;  // zero-initialized int variable defined in the enclosing scope

auto f6 = [j   = i]{ };  // OK, the local j data member is a copy of i.
auto f7 = [&ir = i]{ };  // OK, the local ir data member is an alias to i.
```

Though capturing *by reference* is possible, enforcing `const` on a lambda-capture expression is not:

```
auto f8 = [const i = 10]{ };                      // Error, invalid syntax
auto f9 = [const auto i = 10]{ };                 // Error, invalid syntax
auto fA = [i = static_cast<const int>(10)]{ };  // OK, const is ignored.
```

The initialization expression is evaluated during the *creation* of the closure, not its *invocation*:

```
#include <cassert>  // standard C assert macro

void g()
{
    int i = 0;

    auto fB = [k = ++i]{ };  // ++i is evaluated at creation only.
    assert(i == 1);  // OK

    fB();  // Invoke fB (no change to i).
    assert(i == 1);  // OK
}
```

Finally, using the same identifier as an existing variable is possible for a synthesized capture, resulting in the original variable being **shadowed** (essentially hidden) in the lambda expression's body but not in its **declared interface**. In the example below, we use the (C++11) compile-time operator `decltype` (see "**??**" on page **??**) to infer the C++ type from the initializer in the capture to create a parameter of that same type as that part of its **declared interface**[2,3]:

```
#include <type_traits>  // std::is_same

int i = 0;

auto fC = [i = 'a'](decltype(i) arg)
{
    static_assert(std::is_same<decltype(arg), int>::value, "");
```

[2]Note that, in the shadowing example defining `fC`, GCC version 10.x incorrectly evaluates `decltype(i)` inside the body of the lambda expression as `const char`, rather than `char`; see *Potential Pitfalls: Forwarding an existing variable into a closure always results in an object (never a reference)* on page 60.

[3]Here we are using the (C++14) variable template (see "**??**" on page **??**) version of the standard `is_same` metafunction where `std::is_same<A, B>::value` is replaced with `std::is_same_v<A, B>`.

```
            // i in the interface (same as arg) refers to the int parameter.

    static_assert(std::is_same<decltype(i), char>::value, "");
        // i in the body refers to the char data member deduced at capture.
};
```

Notice that we have again used `decltype`, in conjunction with the standard `is_same` metafunction (which is `true` if and only if its two arguments are the same C++ type). This time, we're using `decltype` to demonstrate that the type (`int`), extracted from the local variable `i` within the declared-interface portion of `fC`, is distinct from the type (`char`) extracted from the `i` within `fC`'s body. In other words, the effect of initializing a variable in the capture portion of the lambda is to hide the name of an existing variable that would otherwise be accessible in the lambda's body.[4]

use-cases-lambdacapture

Use Cases

-objects-into-a-closure

Moving (as opposed to copying) objects into a closure

Lambda-capture expressions can be used to *move* (see "**??**" on page **??**) an existing variable into a closure[5] (as opposed to capturing it *by copy* or *by reference*). As an example of *needing*

[4]Also note that, since the deduced `char` member variable, `i`, is not materially used (**ODR-used**) in the body of the lambda expression assigned (bound) to `fC`, some compilers, e.g., Clang, may warn:

```
warning: lambda capture 'i' is not required to be captured for this use
```

[5]Though possible, it is surprisingly difficult in C++11 to *move* from an existing variable into a closure. Programmers are either forced to pay the price of an unnecessary copy or to employ esoteric and fragile techniques, such as writing a wrapper that hijacks the behavior of its copy constructor to do a *move* instead:

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

template <typename T>
struct MoveOnCopy  // wrapper template used to hijack copy ctor to do move
{
    T d_obj;

    MoveOnCopy(T&& object) : d_obj{std::move(object)} { }
    MoveOnCopy(MoveOnCopy& rhs) : d_obj{std::move(rhs.d_obj)} { }
};

void f()
{
    std::unique_ptr<int> handle{new int(100)};  // move-only
        // Create an example of a handle type with a large body.

    MoveOnCopy<decltype(handle)> wrapper(std::move(handle));
        // Create an instance of a wrapper that moves on copy.

    auto &&lambda = [wrapper](){ /* use wrapper.d_obj */ };
        // Create a "copy" from a wrapper that is captured by value.
}
```

In the example above, we make use of the bespoke ("hacked") `MoveOnCopy` class template to wrap a movable to move from an existing object into a closure, consider the problem of accessing the data managed by **`std::unique_ptr`** (movable but not copyable) from a separate thread — for example, by enqueuing a task in a **thread pool**:

std::unique_ptr

```
ThreadPool::Handle processDatasetAsync(std::unique_ptr<Dataset> dataset)
{
    return getThreadPool().enqueueTask([data = std::move(dataset)]
    {
        return processDataset(data);
    });
}
```

As illustrated above, the `dataset` smart pointer is moved into the closure passed to `enqueueTask` by leveraging lambda-capture expressions — the **`std::unique_ptr`** is *moved* to a different thread because a copy would have not been possible.

e-state-for-a-closure

Providing mutable state for a closure

Lambda-capture expressions can be useful in conjunction with `mutable` lambda expressions to provide an initial state that will change across invocations of the closure. Consider, for instance, the task of logging how many TCP packets have been received on a socket (e.g., for debugging or monitoring purposes)[6]:

std::cout

```
void listen()
{
    TcpSocket tcpSocket(27015);  // some well-known port number
    tcpSocket.onPacketReceived([counter = 0]() mutable
    {
        std::cout << "Received " << ++counter << " packet(s)\n";
        // ...
    });
}
```

Use of `counter = 0` as part of the **lambda introducer** tersely produces a **function object** that has an internal counter (initialized with zero), which is incremented on every received packet. Compared to, say, capturing a `counter` variable *by reference* in the closure, the solution above limits the scope of `counter` to the body of the lambda expression and ties its lifetime to the closure itself, thereby preventing any risk of dangling references.

isting-const-variable

Capturing a modifiable copy of an existing `const` variable

Capturing a variable *by value* in C++11 does allow the programmer to control its `const` qualification; the generated closure data member will have the same `const` qualification as the captured variable, irrespective of whether the lambda is decorated with `mutable`:

object; when the lambda-capture expression tries to *copy* the wrapper (*by value*), the wrapper in turn *moves* the wrapped `handle` into the body of the closure.

[6]In this example, we are making use of the (C++11) `mutable` feature of lambdas to enable the counter to be modified on each invocation.

```
   std::is_same
void f()
{
    int i = 0;
    const int ci = 0;

    auto lc = [i, ci]             // This lambda is not decorated with mutable.
    {
        static_assert(std::is_same<decltype(i), int>::value, "");
        static_assert(std::is_same<decltype(ci), const int>::value, "");
    };

    auto lm = [i, ci]() mutable       // Decorating with mutable has no effect.
    {
        static_assert(std::is_same<decltype(i), int>::value, "");
        static_assert(std::is_same<decltype(ci), const int>::value, "");
    };
}
```

In some cases, however, a lambda capturing a `const` variable *by value* might need to modify that value when invoked. As an example, consider the task of comparing the output of two Sudoku-solving algorithms, executed in parallel:

```
template <typename Algorithm> void solve(Puzzle&);
    // This solve function template mutates a Sudoku grid in place to solution.

void performAlgorithmComparison()
{
    const Puzzle puzzle = generateRandomSudokuPuzzle();
        // const-correct: puzzle is not going to be mutated after being
        // randomly generated.

    auto task0 = getThreadPool().enqueueTask([puzzle]() mutable
    {
        solve<NaiveAlgorithm>(puzzle);  // Error, puzzle is const-qualified.
        return puzzle;
    });

    auto task1 = getThreadPool().enqueueTask([puzzle]() mutable
    {
        solve<FastAlgorithm>(puzzle);  // Error, puzzle is const-qualified.
        return puzzle;
    });

    waitForCompletion(task0, task1);
    // ...
}
```

The code above will fail to compile as capturing `puzzle` will result in a `const`-qualified closure data member, despite the presence of `mutable`. A convenient workaround is to use a (C++14) lambda-capture expression in which a local modifiable copy is deduced:

```
void performAlgorithmComparison2()
{
    // ...

    const Puzzle puzzle = generateRandomSudokuPuzzle();

    auto task0 = getThreadPool().enqueueTask([p = puzzle]() mutable
    {
        solve<NaiveAlgorithm>(p);  // OK, p is now modifiable.
        return p;
    });

    // ...
}
```

Note that use of `p = puzzle` (above) is roughly equivalent to the creation of a new variable using `auto` (i.e., `auto p = puzzle;`), which guarantees that the type of `p` will be deduced as a non-`const` `Puzzle`. Capturing an existing `const` variable as a mutable copy is possible, but doing the opposite is not easy; see *Annoyances: There's no easy way to synthesize a* `const` *data member* on page 61.

itfalls-lambdacapture

Potential Pitfalls

t-(never-a-reference)

Forwarding an existing variable into a closure always results in an object (never a reference)

Lambda-capture expressions allow existing variables to be **perfectly forwarded** (see "**??**" on page **??**) into a closure:

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

template <typename T>
void f(T&& x)  // x is of type forwarding reference to T.
{
    auto lambda = [y = std::forward<T>(x)]
        // Perfectly forward x into the closure.
    {
        // ... (use y directly in this lambda body)
    };
}
```

Because `std::forward<T>` can evaluate to a reference (depending on the nature of `T`), programmers might incorrectly assume that a capture such as `y = std::forward<T>(x)` (above) is somehow either a capture *by value* or a capture *by reference*, depending on the original **value category** of `x`.

Remembering that lambda-capture expressions work similarly to `auto` type deduction for variables, however, reveals that such captures will *always* result in an object, *never* a reference:

```
// pseudocode (auto is not allowed in a lambda introducer.)
auto lambda = [auto y = std::forward<T>(x)] { };
    // The capture expression above is semantically similar to an auto
    // (deduced-type) variable.
```

If `x` was originally an *lvalue*, then `y` will be equivalent to a *by-copy* capture of `x`. Otherwise, `y` will be equivalent to a *by-move* capture of `x`.[7]

If the desired semantics are to capture `x` *by move* if it originated from **rvalue** and *by reference* otherwise, then the use of an extra layer of indirection (using, e.g., `std::tuple`) is required:

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

template <typename T>
void f(T&& x)
{
    auto lambda = [y = std::tuple<T>(std::forward<T>(x))]
    {
        // ... (Use std::get<0>(y) instead of y in this lambda body.)
    };
}
```

In the revised code example above, `T` will be an **lvalue reference** if `x` was originally an **lvalue**, resulting in the **synthetization** of a `std::tuple` containing an **lvalue reference**, which — in turn — has semantics equivalent to `x`'s being captured *by reference*. Otherwise, `T` will not be a reference type, and `x` will be *moved* into the closure.

Annoyances

nnoyances-lambdacapture

There's no easy way to synthesize a `const` data member

ize-a-const-data-member

Consider the (hypothetical) case where the programmer desires to capture a copy of a non-`const` integer `k` as a `const` closure data member:

```
void test1()
{
    int k;
    [k = static_cast<const int>(k)]() mutable  // const is ignored
    {
        ++k;  // "OK" -- i.e., compiles anyway even though we don't want it to
    };
}

void test2()
{
    int k;
    [const k = k]() mutable  // Error, invalid syntax
    {
        ++k;  // no easy way to force this variable to be const
```

[7]Note that both *by-copy* and *by-move* capture communicate **value** for **value-semantic types**.

```
    };
}
```

The language simply does not provide a convenient mechanism for synthesizing, from a modifiable variable, a `const` data member. If such a `const` data member somehow proves to be necessary, we can either create a `ConstWrapper struct` (that adds `const` to the captured object) or write a full-fledged **function object** in lieu of the leaner **lambda expression**. Alternatively, a `const` copy of the object can be captured with traditional (C++11) lambda-capture expressions:

```
int test3()
{
    int k;
    const int kc = k;

    auto l = [kc]() mutable
    {
        ++kc;  // Error, increment of read-only variable kc
    };
}
```

able-callable-objects

`std::function` supports only copyable callable objects

Any lambda expression capturing a move-only object produces a closure type that is itself movable but *not* copyable:

std::unique_ptrstd::move

```
void f()
{
    std::unique_ptr<int> moo(new int);    // some move-only object
    auto la = [moo = std::move(moo)]{ };  // lambda that does move capture

    static_assert(false == std::is_copy_constructible<decltype(la)>::value, "");
    static_assert( true == std::is_move_constructible<decltype(la)>::value, "");
}
```

Lambdas are sometimes used to initialize instances of `std::function`, which requires the stored **callable object** to be copyable:

```
std::function<void()> f = la;  // Error, la must be copyable.
```

Such a limitation — which is more likely to be encountered when using lambda-capture expressions — can make `std::function` unsuitable for use cases where move-only closures might conceivably be reasonable. Possible workarounds include (1) using a different type-erased, **callable object** wrapper type that supports move-only callable objects,[8] (2) taking a performance hit by wrapping the desired **callable object** into a copyable wrapper (such as `std::shared_ptr`), or (3) designing software such that noncopyable objects, once constructed, never need to move.[9]

[8]The `any_invocable` library type, proposed for C++23, is an example of a type-erased wrapper for move-only callable objects; see **calabrese20**.

[9]For an in-depth discussion of how large systems can benefit from a design that embraces local arena

see-also

See Also

- "Lambdas" on page 4 — provides the needed background for understanding the feature in general
- "**??**" on page **??** — illustrates one possible way of initializing the captures
- "**??**" on page **??** — offers a model with the same type deduction rules
- "**??**" on page **??** — gives a full description of an important feature used in conjunction with movable types.
- "**??**" on page **??** — describes a feature that contributes to a source of misunderstanding of this feature

further-reading

Further Reading

None so far

memory allocators and, thus, minimizes the use of moves across natural memory boundaries identified throughout the system, see **lakos22**.

Chapter 3

Unsafe Features

ch-unsafe
sec-unsafe-cpp11 Intro text should be here.

sec-unsafe-cpp14