Chapter 1

Safe Features

ch-safe

sec-safe-cpp11 Intro text should be here.

Chapter 1 Safe Features

sec-safe-cpp14

Chapter 2

Conditionally Safe Features

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

sec-conditional-cpp14

Lambdas Having a Templated Call Operator

genericlambda

C++14 extends the **lambda expression** syntax of C++11 to allow a *templated* definition of the function-call operator belonging to the **closure type**.

Description

description

Generic lambdas are a C++14 extension of C++11 lambda expressions (see .“??” on page ??[**AUs: We have Lambdas or Lambda Captures. There is no Lambda Expressions**]). The relationship between a generic lambda and a nongeneric lambda expression is the same as that between a function template and a nontemplate function: The argument types are deduced at the point of invocation rather than at definition.

Consider two lambda expressions, each of which simply returns its argument:

```
auto identityInt = [](int  a) { return a; };  // nongeneric lambda
auto identity =    [](auto a) { return a; };  // generic lambda
```

Generic lambdas are characterized by the presence of one or more **`auto`** parameters, accepting arguments of any type. In the example above, the first version is a nongeneric lambda having a parameter of concrete type **`int`**. The second version is a generic lambda because its parameter uses the placeholder type **`auto`**. Unlike `identityInt`, which is callable only for **`int`** arguments, `identity` can be applied to any type that can be passed by value:

```
int         a1 = identityInt(42);    // OK, a1 == 42
double      a2 = identityInt(3.14);  // Bug, a2 == 3, truncation warning
const char* a3 = identityInt("hi");  // Error, cannot pass "hi" as int
int         a4 = identity(42);       // OK, a4 == 42
double      a5 = identity(3.14);     // OK, a5 == 3.14
const char* a6 = identity("hi");     // OK, strcmp(a6, "hi") == 0
```

Generic lambdas accomplish this compile-time polymorphism by defining their function call operator — **`operator()`** — as a *template*. Recall that the result of a lambda expression is a **closure object**, an object of unique type having a function call operator; i.e., the closure type is a unique **functor** class. The arguments defined in the lambda expression become the arguments to the function call operator. The following code transformation is roughly equivalent to the definitions of the `identityInt` and `identity` closure objects from the example above:

```
struct __lambda_1  // compiler-generated name; not visible to the user
{
    int operator()(int x) const { return x; }
    // ...
};

struct __lambda_2  // compiler-generated name; not visible to the user
{
    template <typename __T>
    __T operator()(__T x) const { return x; }
    // ...
};

__lambda_1 identityInt = __lambda_1();
__lambda_2 identity    = __lambda_2();
```

Note that the names `__lambda_1`, `__lambda_2`, and `__T` are for only descriptive purposes and are not available to the user; the compiler may choose any name or no name for these entities.

A generic lambda is any lambda expression having one or more parameters declared using the placeholder type **`auto`**. The compiler generates a template parameter type for each **`auto`** parameter in the generic lambda and that type is substituted for **`auto`** in the function call operator's parameter list. In the `identity` example above, **`auto`** `x` is replaced with `__T x`, where `__T` is a new template parameter type. When user code subsequently calls, e.g., `identity(42)`, normal template type deduction takes place and **`operator()<int>`** is instantiated.

Lambda capture and mutable closures

-and-mutable-closures

The closure type produced by a generic lambda is not a class template. Rather, its function call operator and its conversion-to-function-pointer operator (as we'll see later in *Conversion to a pointer to function* on page 10) are function templates. In particular, the **lambda capture**, which creates member variables within the closure type, is treated no differently than any other lambda expression. Similarly, the **`mutable`** qualifier has the same effect for generic lambdas as for nongeneric lambdas:

```
#include <algorithm>  // std::for_each
#include <iterator>   // std::next

template <typename FwdIter>
auto secondBiggest(FwdIter begin, FwdIter end)
    // Return the second-largest element in the range [begin, end),
    // assuming at least two elements and that all values in the range
    // are distinct.
{
    auto ret = *std::next(begin);  // Set to 2nd element
    std::for_each(begin, end,
        [biggest = *begin, &ret](const auto& element) mutable
        {
            if (element < biggest) { return; }
            ret = biggest;
            biggest = element;
        }
    );

    return ret;
}
```

The `ret` declaration uses the placeholder **`auto`** (see Section 2.1.“**??**” on page **??**) to deduce the variable's type to be `*std::next(begin)`, i.e., the element type for the input range.

The return type of `secondBiggest` is also declared **`auto`** and is deduced from the type of `ret` (see Section 3.2."??" on page ??). The generic lambda being passed to `std::for_each` uses the C++14 initialized capture (see Section 2.2."??" on page ??) to initialize `biggest` to the largest value known so far. Because the lambda is declared **`mutable`**, it can update `biggest` each time a larger element is encountered. The `ret` variable is also captured — by reference — and is updated with the previous biggest value when a new biggest value is encountered. Note that, at the point where `ret` appears in the lambda capture, its type has already been deduced. When `for_each` invokes the function call operator, the type of the **`auto`** parameter `element` is conveniently deduced to be the element type for the input range and is thus the same reference type as `ret` except with an added **`const`** qualifier.

Constraints on deduced parameters

s-on-deduced-parameters

A generic lambda may accept any mix of **`auto`** and non**`auto`** parameters:

```
void g1()
{
    auto y1 = [](auto& a, int b, auto c) { a += b * c; };

    int    i = 5;
    double d = 1;

    y1(i, 2, 2);    // i is now 9
    y1(d, 3, 0.5);  // d is now 2.5
}
```

If the **`auto`** placeholder in a generic lambda parameter is part of a type declaration that forms a potentially cv-qualified reference, pointer, pointer-to-member, pointer-to-function, or reference-to-function type, then the allowable arguments will be restricted accordingly:

```
struct C1 { double d_i; };
double f1(int i);

auto y1 = [](const auto& r) { };  // match anything (read only)
auto y2 = [](auto&& r)      { };  //   "   anything (forwarding reference)
auto y3 = [](auto& r)       { };  //   "   only lvalues
auto y4 = [](auto* p)       { };  //   "     "  pointers
auto y5 = [](auto(*p)(int)) { };  //   "     "      "    to functions
auto y6 = [](auto C1::* pm) { };  //   "     "      "    to data members of C1

void g2()
{
    int       i1 = 0;
    const int i2 = 1;

    y1(i1);        // OK, r has type const int&
    y2(i1);        // OK, r has type int&

    y3(5);         // Error, argument is not an lvalue
    y3(i1);        // OK, r has type int&
    y3(i2);        // OK, r has type const int&

    y4(i2);        // Error, i2 is not a pointer
    y4(&i2);       // OK, p has type const int*

    y5(&f1);       // OK, p has type double (*)(int)

    y6(&C1::d_i);  // OK, pm has type double C1::*
}
```

To understand how `y1` and `y2` match any argument type, recall that **`auto`** is a placeholder for a template type argument, say, `__T`. As usual, **`const`** `__T& r` can bind to a **`const`** or non**`const`** *lvalue* or a temporary value created from an *rvalue*. The argument `__T&& r` is a forwarding reference (see Section 2.1."**??**" on page **??**); `__T` will be deduced to an *rvalue* if the argument is an *rvalue* and to an *lvalue* reference otherwise. Because the parameter type for `r` is unnamed — we invented the name `__T` for descriptive purposes only — we must use **`decltype`**`(r)` to refer to the deduced type:

```
#include <utility>  // std::move, std::forward
#include <cassert>  // standard C assert macro

struct C2
{
    int d_value;

    explicit C2(int i)     : d_value(i)                { }
    C2(const C2& original) : d_value(original.d_value) { }
    C2(C2&& other)         : d_value(other.d_value)    { other.d_value = 99; }
};

void g3()
{
    auto y1 = [](const auto& a) { C2 v(a); };
    auto y2 = [](auto&&      a) { C2 v(std::forward<decltype(a)>(a)); };

    C2 a(1);

    y1(a);            assert(1  == a.d_value);  // Copies from a
    y1(std::move(a)); assert(1  == a.d_value);  //    "     "  a
    y2(a);            assert(1  == a.d_value);  //    "     "  a
    y2(std::move(a)); assert(99 == a.d_value);  // Moves    "  a
}
```

In this example, `y1` always invokes the copy constructor for `C2` because `a` has type **`const`** `C2&` regardless of whether we instantiate it with an *lvalue* or *rvalue* reference to `C2`. Conversely, `y2` forwards the value category of its argument to the `C2` constructor using `std::forward` according to the common idiom for forwarding references. If passed an *lvalue* reference, the copy constructor is invoked; otherwise, the move constructor is invoked. We can tell the difference because `C2` has a move constructor that puts the special value `99` into the moved-from object.

The **auto** placeholder in a generic lambda parameter cannot be a type argument in a template specialization, a parameter type in the prototype of a function reference or function pointer, or the class type in a pointer to member[1]:

```
#include <vector>  // std::vector
auto y7 = [](const std::vector<auto>& x) { };  // Error, invalid use of auto
auto y8 = [](double (*f)(auto)) { };           // Error,    "      "   "    "
auto y9 = [](int auto::* m) { };               // Error,    "      "   "    "
```

Because of this restriction, there are no contexts where more than one **auto** is allowed to appear in the declaration of a single lambda parameter. Template parameters *are* allowed in these contexts for regular function templates, so generic lambdas are less expressive than handwritten functor objects in this respect:

```
struct manualY7
{
    template <typename T>
    void operator()(const std::vector<T>& x) const { }  // OK, can deduce T
};

struct manualY8
{
    template <typename T>
    void operator()(double (*f)(T)) const { }  // OK, can deduce T
};

struct manualY9
{
    template <typename R, typename T>
    void operator()(R T::* m) const { }  // OK, can deduce R and T
};
```

Not only do `manualY7`, `manualY8`, and `manualY9` successfully deduce `T` where `y7`, `y8`, and `y9` could not deduce **auto**, `manualY9` deduces two template arguments for a single function parameter. There is a trade-off between the benefits of lambda expressions, such as defining the function in place at the point of use, and the pattern-matching power of manually written function templates. See Annoyances — Cannot use full power of template-argument deduction.

A default value on an **auto** parameter, while allowed, is not useful because it defaults only the *value* and not the *type* of the template parameter. Invocation of such a generic lambda requires the programmer to either supply a value for the argument, which defeats the point of a defaulted argument, or explicitly instantiate **operator()**, which is gratuitously awkward:

```
void g1()
{
    auto y = [](auto a = 3) { return a * 2; };
    y(5);                // OK, returns 10
```

[1]GCC 10.2 does allow **auto** in both template arguments and function prototype parameters and deduces the template parameter type in the same way as for a regular function template. MSVC 19.28 allows **auto** in the parameter list for a function reference or function pointer but not in the other two contexts.

```
    y();                  // Error, cannot deduce type for parameter a
    y.operator()<int>(); // OK, returns 6
}
```

iadic-generic-lambdas

Variadic generic lambdas

If a placeholder argument to a generic lambda is followed by an ellipsis (...), then the parameter becomes a variadic parameter pack and the function call operator becomes a variadic function template; see Section 2.1."**??**" on page **??**:

```
#include <tuple>  // std::tuple, std::make_tuple
auto y11 = [](int i, auto&&... args)
{
    return std::make_tuple(i, std::forward<decltype(args)>(args)...);
};

std::tuple<int, const char*, double> tpl1 = y11(3, "hello", 1.2);
```

The `y11` closure object forwards all of its arguments to `std::make_tuple`. The first argument must have type convertible to **`int`**, but the remaining arguments can have any type. Assuming the invented name `__T` for the template type parameter, the generated function call operator for `y11` would have a variadic template parameter list:

```
struct __lambda_3
{
    template <typename... __T> void operator()(int, __T&&... args) const
    {
        /* ... */
    }
};
```

The standard limitations on variadic function templates apply. For example, only a variadic parameter pack at the end of the argument list will match function-call arguments at the point of invocation. In addition, because **`auto`** is not permitted in a template-specialization parameter, the usual methods of defining function templates with multiple variadic parameter packs do not work for generic lambdas:

```
// Attempt to define a lambda expression with two variadic parameter packs.
auto y12 = [](std::tuple<auto...>&, auto...args) { };
    // Error, auto is a template argument in tuple specialization.
```

a-pointer-to-function

Conversion to a pointer to function

A nongeneric lambda expression with an empty lambda capture can be converted implicitly to a function pointer with the same signature. A generic lambda with an empty lambda capture can similarly be converted to a regular function pointer, where the parameters in the prototype of the target pointer type drive deduction of the appropriate **`auto`** parameters in the generic lambda signature:

```
auto y1 = [](int a, char b) { return a; };   // nongeneric lambda
```

```
int (*f1)(int, char) = y1;                    // OK, conversion to pointer

auto y2 = [](auto a, auto b) { return a; };  // generic lambda
int    (*f2)(int, int)    = y2;  // OK, instantiates operator()<int, int>
double (*f3)(double, int) = y2;  // OK, instantiates operator()<double, int>
char   (*f4)(int, char)   = y2;  // Error, incorrect return type
```

If the function target pointer is a variable template (see Section 1.2."**??**" on page **??**), then the deduction of the arguments is delayed until the variable template itself is instantiated:

```
template <typename T> int (*f5)(int, T) = y2;  // variable template
int (*f6)(int, short) = f5<short>;             // instantiate f5
```

Each function pointer is produced by calling a conversion operator on the closure object. In the case of the generic lambda, the conversion operator is also a template. Template-argument deduction and return-type deduction are performed on the conversion operator, then the conversion operator instantiates the function call operator. Intuitively, it is as though the function call operator were a static member function template of the closure and the conversion operator returned a pointer to it.

use-cases

Use Cases

able-lambda-expressions

Reusable lambda expressions

One of the benefits of lambda expressions is that they can be defined within a function, close to the point of use. Saving a lambda expression in a variable allows it to be reused within the function. This reusability is greater for generic lambdas than for nongeneric lambdas, just as function templates are more reusable than ordinary functions. Consider, for example, a function that partitions a vector of strings and a vector of vectors based on the length of each element:

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

void partitionByLength(std::size_t                    pivotLen,
                       std::vector<std::vector<int>>& v1,
                       std::vector<std::string>&      v2)
{
    auto condition = [pivotLen](const auto& e) { return e.size() < pivotLen; };

    std::partition(v1.begin(), v1.end(), condition);
    std::partition(v2.begin(), v2.end(), condition);
}
```

The `condition` generic lambda can be used to partition both `vector`s because its function call operator can be instantiated on either element type. The capture of `pivotLen` is performed only once, when the lambda expression is evaluated to yield `condition`.

Applying a lambda to each element of a tuple

ch-element-of-a-tuple

An `std::tuple` is a collection of objects having heterogeneous types. We can apply a functor to each element of a `tuple` by using some metaprogramming features of the C++14 Standard Library:

```
#include <utility>  // std:::index_sequence, std::make_index_sequence
#include <tuple>    // std::tuple, std::tuple_size, std::get

template <typename Tpl, typename F, std::size_t... I>
void visitTupleImpl(Tpl& t, F& f, std::index_sequence<I...>)
{
    auto discard = { (f(std::get<I>(t)), 0)... };
}

template <typename Tpl, typename F>
void visitTuple(Tpl& t, F f)
{
    visitTupleImpl(t, f,
                   std::make_index_sequence<std::tuple_size<Tpl>::value>());
}
```

The `visitTuple` function uses `make_index_sequence` to generate a compile-time pack of sequential indexes, from 0 up to, but not including, the number of elements in the `tuple`-like argument `t`. The original arguments, along with this pack of indexes, are passed to `visitTupleImpl`, which applies each index to `t` and then calls the functor `f` on the resulting element.[2] The implementation uses an idiom that discards the sequence of return values (if any) from the calls to `f`: Regardless of the type of value returned by `f(std::get<I>(t))`, even if it's **`void`**, the comma expression `(f(std::get<I>(t)), 0)` always yields the integer `0`. The order of evaluation of elements in an `initializer_list` is guaranteed left to right. The result of this expansion is an `initializer_list` used to initialize `discard`, which, as its name suggests, is then discarded.

Once we have a `visitTuple` function, we can use it to apply a generic lambda to the elements of a `tuple`:

```
#include <ostream>  // std::ostream, std::endl
void f1(std::ostream& os)
{
    std::tuple<int, float, const char*> t{3, 4.5, "six"};
    visitTuple(t, [&os](const auto& v){ os << v << ' '; });
    os << std::endl;
}
```

The first line constructs a `tuple` with three different element types. The second line then calls `visitTuple` to visit each element of `t` and apply a lambda function to it. The lambda capture stores a reference to the output stream, and the **lambda body** prints the current

[2]This pattern for using `make_index_sequence` with `tuple`-like objects is explained in a number of places on the Internet (see **?**), and a basic knowledge of variadic templates Section 2.1."**??**" on page **??** is critical to understanding this idiom.

element to the output stream. This code would not work if the lambda were not generic because the type of v is different for each element in t.

terse,-robust-lambdas

Terse, robust lambdas

Often, function objects are more convenient to write as generic lambdas, rather than non-generic lambdas, even if they are used only once and their genericity is not fully exploited. Consider the case where a lambda's parameters have long, elaborate types:

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

template <typename Iterator>
void f1(Iterator begin, Iterator end)
    // Sort [begin, end) using a nongeneric lambda
{
    std::sort(begin, end,
              [](typename std::iterator_traits<Iterator>::reference a,
                 typename std::iterator_traits<Iterator>::reference b)
              {
                  return a < b;
              });
}
```

The types for parameters a and b of the comparison lambda expression are not easy to type or read. Compare this code with similar code using generic lambdas:

```
template <typename Iterator>
void f2(Iterator begin, Iterator end)
    // Sort [begin, end) using a generic lambda
{
    std::sort(begin, end,
              [](const auto& a, const auto& b) { return a < b; });
}
```

The code is simpler to write and read because it takes advantage of the lambda expression's being defined at the point of use; even though the argument types are not written out, their meaning is still clear. A generic lambda is also more robust when the types involved change, even in a small way, as it adapts to changes in argument types as long as they support the same interface.

recursive-lambdas

Recursive lambdas

Since neither generic nor nongeneric lambda expressions have names, defining a lambda that calls itself recursively is tricky. One way to accomplish recursion is by passing the lambda as a parameter to itself. This technique requires a generic lambda so that it can deduce its own type from its argument:

```
auto fib = [](auto self, int n) -> int  // Compute the nth Fibonacci number.
{
    if (n < 2) { return n; }
    return self(self, n - 1) + self(self, n - 2);
};

int fib7 = fib(fib, 7);  // returns 13
```

Notice that when we invoke the recursive lambda, we pass it as an argument to itself, both to the external call and to the internal recursive calls. To avoid this somewhat awkward interface, a special function object called a **Y Combinator** can be used.[3] The Y Combinator object holds the closure object to be invoked recursively and passes it to itself:

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

template <typename Lambda>
class Y_Combinator {
    Lambda d_lambda;

public:
    Y_Combinator(Lambda&& lambda) : d_lambda(std::move(lambda)) { }

    template <typename... Args>
    decltype(auto) operator()(Args&&...args) const
    {
        return d_lambda(*this, std::forward<Args>(args)...);
    }
};

template <typename Lambda>
Y_Combinator<Lambda> Y(Lambda&& lambda) { return std::move(lambda); }
```

The function call operator for `Y_Combinator` is a **variadic function template** (see Section 2.1."**??**" on page **??**) that passes itself to the stored closure object, `d_lambda`, along with zero or more additional arguments supplied by the caller. Thus, `d_lambda` and the `Y_Combinator` are mutually-recursive functors. The `Y` function template constructs a `Y_Combinator` from a lambda expression.

To use a `Y_Combinator`, pass a recursive generic lambda to `Y`; the resulting object is the one that you would call from code:

```
auto fib2 = Y([](auto self, int n) -> int
{
    if (n < 2) { return n; }
    return self(n - 1) + self(n - 2);
});

int fib8 = fib2(8);  // returns 21
```

Note that the recursive lambda still needs to take `self` as an argument, but, because `self` is a `Y_Combinator`, it does not need to pass `self` to itself. Unfortunately, we must now specify the return type of the lambda because the compiler cannot deduce the return type of the mutually recursive invocations of `self`. The usefulness of a Y Combinator in C++ is

[3] **?**

debatable given alternative, often simpler ways to achieve recursion, including using ordinary function templates instead of lambda expressions.[4]

nditional-instantiation

Conditional instantiation

Because a generic lambda defines a function template that is not instantiated unless it is called, it is possible to put code into a generic lambda that would not compile for certain argument types. We can, thus, selectively instantiate calls to generic lambdas based on some compile-time conditional expression similar to the **`if constexpr`** feature introduced in C++17:

```
#include <iostream>     // std::cout
#include <type_traits>  // std::is_pointer
#include <functional>   // std::reference_wrapper
#include <cassert>      // standard C assert macro

// Identity functor: Each call to operator() returns its argument unchanged.
struct Identity
{
    template <typename T>
    decltype(auto) operator()(T&& x) { return std::forward<T>(x); }
};

template <typename F1, typename F2>
decltype(auto) ifConstexprImpl(std::true_type, F1&& f1, F2&&)
    // call f1, which is the "then" branch of ifConstexpr
{
    return std::forward<F1>(f1)(Identity{});
}

template <typename F1, typename F2>
decltype(auto) ifConstexprImpl(std::false_type, F1&&, F2&& f2)
    // call f2, which is the "else" branch of ifConstexpr
{
    return std::forward<F2>(f2)(Identity{});
}

template <bool Cond, typename F1, typename F2>
decltype(auto) ifConstexpr(F1&& f1, F2&& f2)
    // If the compile-time condition Cond is true, return the result of
    // invoking f1, else return the result of invoking f2.  The
    // invocations of f1 and f2 are each passed an instance of Identity
    // as their sole argument.
{
    using CondT = std::integral_constant<bool, Cond>;
    return ifConstexprImpl(CondT{}, std::forward<F1>(f1), std::forward<F2>(f2));
}
```

[4]Derevenets suggested that `y_combinator` should be part of the C++ Standard Library (see **?**), but the proposal was rejected for addressing a problem deemed not worth solving.

The `Identity` functor class, as its name suggests, returns its argument unchanged, preserving both its type and value category (*lvalue* or *rvalue*). This functor exists solely to be passed to a generic lambda, which will then invoke it as described below.

The two overloads of `ifConstexprImpl` each take two functor arguments, only one of which is used. The first overload invokes only its first argument; the second overload invokes only its second argument. In both cases, an `Identity` object is passed as the only argument to the functor invocation.

The `ifConstexpr` function template, in addition to taking two functor arguments, must be instantiated with an explicit compile-time Boolean value. It calls one of the two `ifConstexprImpl` functions such that its first argument is instantiated and invoked if the condition is **`true`** — the "then" clause of the simulated **`if constexpr`** — and the second argument is instantiated and invoked otherwise — the "else" clause of the simulated **`if constexpr`**.

To see how `ifConstexpr` achieves conditional instantiation, let's write a function that dereferences an object. The object being dereferenced can be a pointer, an `std::reference_wrapper`, or a class that behaves like `reference_wrapper`. We present both a C++17 version that uses **`if constexpr`** and a C++14 version that uses `ifConstexpr`, above:

```
template <typename T>
decltype(auto) objectAt(T ref)
    // Generalized dereference for pointers and reference_wrapper-like
    // objects: If T is a pointer type, return *ref; otherwise, return
    // ref.get().
{
#if __cplusplus >= 201703
    // C++17 version
    if constexpr (std::is_pointer<T>::value) { return *ref; }
    else                                      { return ref.get(); }
#else
    // C++14 version
    return ifConstexpr<std::is_pointer<T>::value>(
        [=](auto dependent) -> decltype(auto) { return *dependent(ref); },
        [=](auto dependent) -> decltype(auto) { return dependent(ref).get(); });
#endif
}
```

Looking at the C++17 version first, we see that if `ref` is a pointer, then we return `*ref`; otherwise, we return `ref.get()`. In both cases, the code would not compile if the branch not taken were instantiated; i.e., `*ref` is ill formed for `std::reference_wrapper`, and `ref.get()` is ill formed for pointers. The code depends on the ill-formed branch being discarded at compile time.

The C++14 version needs to follow a very specific idiom whereby each conditional branch is represented as a generic lambda with one **`auto`** parameter that is expected to be an `Identity` object. Because it is a template parameter, `dependent` is a **dependent type**. In C++ template parlance, a dependent type is one that cannot be known until template instantiation. Critically, the compiler will not check semantic correctness of an expression containing a value of dependent type unless and until the template is instantiated with a concrete type. Any such expression that uses a value of dependent type also has dependent type. By wrapping `ref` in a call to `dependent`, therefore, we ensure that the compiler will not test if `*ref` or `ref.get()` is valid until the generic lambda is instantiated. Our implementation of `ifConstexpr` ensures that only the *correct* lambda is instantiated, thus preventing a compiler error.[5]

A simple test for `objectAt` instantiates it with both pointer and `reference_wrapper` arguments and verifies that the value returned has both the correct value and address:

```
void f1()
{
    int i = 8;

    int&       i1 = objectAt(&i);
    int&       i2 = objectAt(std::reference_wrapper<int>(i));
    const int& i3 = objectAt(std::reference_wrapper<const int>(i));

    assert(8 == i1);
    assert(&i1 == &i);
    assert(&i2 == &i);
    assert(&i3 == &i);
}
```

potential-pitfalls

Potential Pitfalls

Although the generic feature is generally safe, generic lambdas are an incremental advance over regular lambda expressions. All pitfalls of the nongeneric feature apply to this feature as well; see Section 2.1."??" on page ??.

annoyances

Annoyances

late-argument-deduction

Cannot use full power of template-argument deduction

A normal function template can constrain template parameters to patterns that include argument prototypes and template instantiations:

```
#include <vector>  // std::vector
template <typename T> void f1(std::vector<T>& v) { }  // T is the element type
```

Not only is the argument `v` constrained to being a `vector`, but the deduced element type `T` is available for use within the function. The same sort of pattern matching is not available portably for generic lambdas:

```
auto y1 = [](std::vector<auto>& v) { };  // Error, auto as template parameter
```

Constraining an **`auto`** parameter using metaprogramming, e.g., through the use of `std::enable_if`, is sometimes possible:

```
#include <type_traits>  // std::enable_if_t, std::is_same,
                        // std::remove_reference_t
auto y2 = [](auto& v) -> std::enable_if_t<
```

[5] Paul Fultz suggested the idea for using an identity functor to mark expressions as having dependent type (?).

```
    std::is_same<
        std::vector<typename std::remove_reference_t<decltype(v)>::value_type>&,
        decltype(v)
    >::value> { };
```

The `y2` closure can be called only with a vector. Any other type will fail substitution because `is_same` will return `false` if substitution even gets that far; substitution might fail earlier if the type for `v` does not have a nested `value_type`. Passing non`vector` arguments to this constrained lambda will now fail at the call site, rather than, presumably, failing during instantiation of `y2(v)`:

```
void g1()
{
    int                i;
    std::vector<int>   v1;
    std::vector<float> v2;

    y2(i);   // Error, cannot call y2 on a nonvector
    y2(v1);  // OK, v1 is a vector
    y2(v2);  // OK, v2 is a vector
}
```

For all of the additional complication on `y2`, the element type for our `vector` is still not available within the lambda body, as it was for the function body for `f1`, above; we would need to repeat the type name `typename std::remove_reference_t<decltype(v)>::value_type` if the element type became necessary.

This annoyance is mitigated, however, because lambda expressions cannot be overloaded. In the absence of overloading, there is little benefit to removing a call from the overload set compared to simply letting the instantiation fail, especially as most lambda expressions are defined at the point of use, making it comparatively easy to diagnose a compilation problem if one occurs. Moreover, this point-of-use definition is already tuned to its expected use case, so constraints are often redundant, adding little additional safety to the code.

Difficulty in constraining multiple arguments

ng-multiple-arguments

Often, we want to restrict function template arguments such that two or more arguments have related types. For example, an operation might require two iterators of the same type or one argument to be a pointer to the type of the other argument. Generic lambdas provide only limited support for interargument patterns. Each parameter to a generic lambda that contains **auto** is completely independent of the other parameters, so the normal mechanism used for function templates — using the same named type parameter (e.g., `T` or `U`) in multiple places — is unavailable:

```
void g1()
{
    auto y = [](auto a, auto b) { };  // No interargument constraints
    y(1, 2);           // OK, arguments of type int and int
    y("one", "two");  // OK,     "      "   "  const char* and const char*
    y(1, "two");      // OK,     "      "   "  int and const char*
}
```

Limited constraints are possible — e.g., requiring that arguments have the same type — by using the **decltype** operator to declare later parameters based on the types of earlier parameters; see Section 1.1."**??**" on page **??**:

```
void g2()
{
    auto y = [](auto a, decltype(a) b) { };  // a and b have the same type.
    y(1, 2);          // OK, both arguments are of type int.
    y("one", "two");  // OK,   "      "      "   "   "  const char*.
    y(1, "two");      // Error, mixed argument types int and const char*
}
```

If the relationship between the parameter types is even slightly different than an exact match, expressing it can become complicated. For example, if parameter `a` is a pointer and we want parameter `b` to be a value of the type pointed to by `a`, our first approach might be to try `decltype(*a)`:

```
int i = 0;

void g3()
{
    auto y = [](auto* a, decltype(*a) b) { *a = b; };
    y(&i, 5);  // Error, can't bind rvalue 5 to int& b
}
```

Unfortunately, as we see, `decltype(*a)` yields a non**const** *lvalue* reference, which cannot bind to the *rvalue* `5`. Our next attempt is to **const**-qualify `b`, since **const** references *can* bind to *rvalues*:

```
void g4()
{
    auto y = [](auto* a, const decltype(*a) b) { *a = b; };
    y(&i, 5);  // Error, const applied to reference is ignored.
}
```

This approach fails because **const** can be applied only to the *referred-to* type; applying **const** to the reference has no effect. If we use the standard metafunction `std::remove_reference_t`, we can finally get to the type to which `a` points, without any reference specifiers:

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

void g5()
{
    auto y = [](auto* a, std::remove_reference_t<decltype(*a)> b)
    {
        *a = b;
    };
    y(&i, 5); // OK, pass 5 by value for int b.
}
```

Note that `y` takes argument `b` by value, which can be inefficient for types having an expensive copy constructor. When the type of an argument is unknown, we usually pass it by **const** reference:

```
void g6()
{
    auto y = [](auto* a, const std::remove_reference_t<decltype(*a)>& b)
    {
        *a = b;
    };
    y(&i, 5); // OK, bind 5 to const int& b.
}
```

In the last example, parameter `b` is always a **`const`** *lvalue* reference, never an *rvalue* reference. When constraining a type this way, we forfeit the possibility of perfect forwarding.

see-also

See Also

- "??" (§1.1, p. ??) ♦ provides the only way to name the type of an **`auto`** parameter in a generic lambda.
- "??" (§2.1, p. ??) ♦ allows a lambda expression, which has unnamed type, to be saved in a variable and also details the rules for deducing an **`auto`** parameter from its argument expression.
- "??" (§2.1, p. ??) ♦ provide the most general way to declare a generic lambda's parameters, preserving both the parameters' types and their value categories.
- "??" (§2.1, p. ??) ♦ introduced the facility for locally defined anonymous function objects that generic lambdas expand on.
- "??" (§2.1, p. ??) ♦ allow a template, including a generic lambda, to have a variable number of parameters.
- "??" (§2.2, p. ??) ♦ describes the initialized capture syntax added to lambda captures in C++14.
- "??" (§3.2, p. ??) ♦ describes how a function, including the function call operator for a lambda expression, can deduce its return type from its **`return`** statements.

further-reading

Further Reading

- ?
- ?
- ?

Chapter 3

Unsafe Features

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

Chapter 3 Unsafe Features

sec-unsafe-cpp14