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.

sec-conditional-cpp14

Relaxed Restrictions on constexpr Functions

-constexpr-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 in C++11/14: 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 6.

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 permitted:

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

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)
```

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 man-

[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.

ner. 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. *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 run-time evaluation, thereby requiring programmers to provide and maintain *two* separate

---

[3]For a more efficient (yet less intuitive) C++11 algorithm, see *Appendix: Optimized C++11 Example Algorithms, Recursive Fibonacci* on page 12.

[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?
```

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 nth 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.

rogramming-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`):

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

template <typename = Nil, typename = Nil, typename = Nil, typename = Nil>
struct TypeList { };
```

```
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, double, Nil> listOfThreeIntDoubleNil;
```

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]:

axedconstexpr-countcode

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

```
        // 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:

```
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

// ...
```

```
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:

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

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

```
        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.
- "??" — 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.

[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 9) — 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.

further-reading

Further Reading

None so far

11-example-algorithms

Appendix: Optimized C++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;
}
```

elist-count-algorithm

constexpr type list 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

a-capture-expressions

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

Description

description

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

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

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):

[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.

```
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_v<decltype(arg), int>, "");
         // i in the interface (same as arg) refers to the int parameter.

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

---

[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 19.

[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>`.

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]

e-cases-lambdacapture

Use Cases

ojects-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* 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**:

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

[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:

```
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 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.

```
    {
        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.

ble-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]:

```
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.

existing-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`:

```
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_v<decltype(i), int>, "");
        static_assert(std::is_same_v<decltype(ci), const int>, "");
    };

    auto lm = [i, ci]() mutable       // Decorating with mutable has no effect.
    {
```

[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.

```
        static_assert(std::is_same_v<decltype(i), int>, "");
        static_assert(std::is_same_v<decltype(ci), const int>, "");
    };
}
```

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:

```
// ...

const Puzzle puzzle = generateRandomSudokuPuzzle();

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

// ...
```

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 20.

-pitfalls-lambdacapture

Potential Pitfalls

ect-(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:

```
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:

```
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.)
```

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

```
    };
}
```

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

oyances-lambdacapture

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

e-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:

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

[const k = k]() mutable  // error: invalid syntax
{
    ++k;  // no easy way to force this variable to be const
};
```

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 k;
const int kc = k;

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

`std::function` supports only copyable callable objects

able-callable-objects

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

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

    static_assert(false == std::is_copy_constructible_v<decltype(la)>, "");
```

```
    static_assert( true == std::is_move_constructible_v<decltype(la)>, "");
}
```

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]

see-also

See Also

- "**??**" on page **??** — 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

[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 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