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.

Range-Based for Loops

range-based-forloops rangefor

Range-Based **for** loops provide a simplified, more compact, syntax for iterating through a range of elements.

Description

description-rangefor

Iterating over the elements of a collection is a fundamental operation usually performed with a **for** loop:

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

void f1(const std::vector<std::string>& vec)
{
    for (std::vector<std::string>::const_iterator i = vec.begin();
         i != vec.end(); ++i)
    {
        // ...
    }
}
```

The code above iterates over the strings in a `std::vector`. It is significantly more verbose than similar code in other languages because it uses a very general-purpose construct, the **for** loop, to perform the specialized but common task of traversing a collection. In C++11, the definition of `i` can be simplified somewhat by using **auto**:

```
void f2(const std::vector<std::string>& vec)
{
    for (auto i = vec.begin(); i != vec.end(); ++i)
    {
        const std::string& s = *i;
        // ...
    }
}
```

Although **auto** does have a number of potential pitfalls, this use of **auto** to deduce the return type of `vec.begin()` is one of the safer idiomatic uses; see Section 2.1."**??**" on page **??**. While this version of the loop is simpler to write, it still uses the fully general, three-part **for** construct. Moreover, it evaluates `vec.end()` each time through the loop.

The C++11 **ranged-based for loop** (sometimes colloquially referred to as the "foreach" loop) is a more concise loop notation tuned for traversing the elements of a container or other sequential range. A ranged-based **for** loop works with *ranges* and *elements* rather than *iterators* or *indexes*:

-range-based-for-loop

```
void f3(const std::vector<std::string>& vec)
{
    for (const std::string& s : vec)
    {
```

```
        // ...
    }
}
```

The loop in the above example can be read as "for each element `s` in `vec` ...". There is no need to specify the name or type of the iterator, the loop-termination condition, or the increment clause; the syntax is focused purely on yielding (in order) each element of the collection for processing within the body of the loop. [1]

Specification

specification

The syntax for a ranged-based **for** loop declares a loop variable and specifies a range of elements to be traversed:

```
for ( *for-range-declaration* : *range-expression* ) *statement*
```

The compiler treats this high-level construct as though it were transformed into a lower-level **for** loop with the following pseudocode:

```
{
    auto&& __range = range-expression;
    for (auto __begin = begin-expr, __end = end-expr;
        __begin != __end;
        ++__begin)
    {
        for-range-declaration = *__begin;
        statement
    }
}
```

The variables `__range`, `__begin`, and `__end`, above, are for *exposition only*, i.e., the compiler does not necessarily generate variables with those names and user code is not permitted to access those variables directly.

The `__range` variable is defined as a **forwarding reference** (see Section 2.1."**??**" on page **??**); it will bind to any type of **range expression**, regardless of its **value category** (*lvalue* or *rvalue*). If the range expression yields a temporary object, its lifetime is extended, if necessary, until `__range` goes out of scope. While this **lifetime extension** of temporary objects works in most cases, it is insufficient when `__range` doesn't bind directly to the temporary created by the range expression, potentially resulting in subtle bugs; see Potential PitfallsLifetime of temporaries in the range expression.

The *begin-expr* and *end-expr* expressions used to initialize the `__begin` and `__end` variables, respectively, define a half-open range of elements starting with `*__begin` and including all of the elements in the `__range` up to but not including `*__end`. The precise meaning of *begin-expr* and *end-expr* were clarified in C++14 but were essentially the same in C++11[2]:

[1] In C++20, the syntax has been enhanced slightly to permit an optional leading variable declaration clause, e.g., `for (std::lock_guard g(myMutex); const std::string& s : vec) { /* ... */ }`.

[2] The rules for interpreting *begin-expr* and *end-expr* were slightly unclear in C++11. A defect report, CWG 1442, clarified the wording retroactively. C++14 clarified the wording further.

- If `__range` refers to an array, then *begin-expr* is the address of the first element of the array and *end-expr* is the address of one past the last element of the array.

- If `__range` refers to a class object and `begin` and/or `end` are members of that class, then *begin-expr* is `__range.begin()` and *end-expr* is `__range.end()`. Note that if `begin` or `end` are found in the class, then both of these expressions must be valid or else the program is ill formed.

- Otherwise, *begin-expr* is `begin(__range)` and *end-expr* is `end(__range)`, where `begin` and `end` are found using **argument-dependent lookup (ADL)**. Note that `begin` and `end` are looked up only in the namespaces associated with the expressions; names that are local to the context of the ranged-based **`for`** loop are not considered; see Annoyances.

Thus, a container such as `vector`, with conventional `begin` and `end` member functions, provides everything necessary for a ranged-based **`for`** loop, as we saw in the `f3` example on 4. Note that *end-expr* — `__range.end()` in the case of the `vector` — is evaluated only once, unlike the idiomatic low-level **`for`** loop, where it is evaluated prior to every iteration.

In C++11 and C++14, `__begin` and `__end` are required to have the same type.[3] Although the `__begin` and `__end` variables look and act like iterators, they need not conform to all of the iterator requirements in the Standard. Specifically, the type of `__begin` and `__end` must support prefix **`operator`**`++` but not necessarily postfix **`operator`**`++`, and it must support **`operator`**`!=` but not necessarily **`operator`**`==`.

The *for-range-declaration* declares the loop variable. Any declaration that can be initialized with `*__begin` will work. For instance, if `*__begin` returns a reference to a modifiable object of, e.g., **`int`** type, then **`int`** `j`, **`int`**`& j`, **`const int`**`& j`, and **`long`** `j` would all be valid *for-range-declaration*s declaring a loop variable `j`. Alternatively, the type of the loop variable can be deduced using **`auto`** — i.e., **`auto`** `j`, **`auto`**`& j`, **`const auto`**`& j`, or **`auto`**`&& j` (see Section 2.1."**??**" on page **??**).

The sequence being traversed can be modified through the loop variable only if `__begin` returns a reference to a modifiable type and the loop variable is similarly declared as a reference to a modifiable type (e.g., **`int`**`&`, **`auto`**`&`, or **`auto`**`&&`). Note that the **for-range declaration** must define a *new* variable; unlike a traditional **`for`** loop, it cannot name an existing variable already in scope:

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

void f1(std::vector<int>& vec)
{
    const std::vector<int>& cvec = vec;

    for (auto&& i : cvec)
    {
        i = 0;  // Error, i is a reference to const int.
    }
```

[3] The C++17 Standard changes the defining code transformation of the range-based **`for`** loop so as to allow `__begin` and `__end` to have different types as long as they are comparable using `__begin != __end`.

```
    for (int j : vec)
    {
        j = 0;  // Bug, j is a loop-local variable; vec is not modified.
    }

    for (int& k : vec)
    {
        k = 0;  // OK, set element of vec to 0.
    }

    int m;
    for (     m : vec) { /* ... */ }  // Error, m does not define a variable.
    for (int& m : vec) { /* ... */ }  // OK, loop m hides function-scope m.
}
```

Since `cvec` is **`const`**, the element type returned by `*begin(cvec)` is **`const int`**`&`. Thus, `i` is deduced as **`const int`**`&`, making invalid any attempt to modify an element through `i`. The second loop is valid C++11 code but has a subtle bug: `j` is not a reference — it contains a *copy* of the current element in the `vector` — so modifying `j` has no effect on the vector. The third loop correctly sets all of the elements of `vec` to zero; the loop variable `k` is a reference to the current element, so setting it to zero modifies the original vector. The first `m` loop attempts to re-use local variable `m` as a loop variable; while this would be legal in a traditional **`for`** loop, it is ill formed in a ranged-based **`for`** loop. Finally, the last loop re-uses the name `m` from the surrounding scope, hiding the old name for the duration of the loop, just as it would in a traditional **`for`** loop.

The *statement* that makes up the loop body can contain anything that is valid within a traditional **`for`** loop body. In particular, a **`break`** statement will exit the loop immediately and a **`continue`** statement will skip to the next iteration.

Applying this transformation to the `f3` example (see 4) from the previous section, we can see how the ranged-based **`for`** loop hooks into the iterator idiom to traverse a `vector` of `string` elements:

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

void f3(const std::vector<std::string>& vec)
{
    // for (const std::string& s : vec) { /* ... */ }
    {
        auto&& __range = vec;  // reference to the std::vector
        for (auto __begin = begin(__range), __end = end(__range);
             __begin != __end;
             ++__begin)
        {
            const std::string& s = *__begin;  // Get current string element.
            {
                // ...
            }
```

```
        }
    }
}
```

In this expansion, `__range` has type `const std::vector<std::string>&` while `__begin` and `__end` have type `std::vector<std::string>::const_iterator`.

Traversing arrays and initializer lists

and-initializer-lists

The `<iterator>` standard header defines array overloads for `std::begin` and `std::end` such that, when applied to a C-style array, `arr`, having a known number of elements, `__bound`, `std::begin(arr)` returns the address of the first element of `arr` and `std::end(arr)` returns the address of one past that of the last element of `arr`, i.e., `arr + __bound`. This functionality is built into the initialization of `__begin` and `__end` as a special case, in the expansion of a range-based **`for`** loop, so that it is possible to traverse the elements of an array without needing to `#include <iterator>`:

```
void f1()
{
    double data[] = {1.9, 2.8, 4.7, 7.6, 11.5, 16.4, 22.3, 29.2, 37.1, 46.0};
    for (double& d : data)
    {
        d *= 3.0;  // triple every element in the array
    }
}
```

In the above example, the reference `d` is bound, in turn, to each element of the array. The size of the array is not encoded anywhere in the loop syntax, either as a literal or as a symbolic value, simplifying the specification of the loop and preventing errors. Note that only arrays whose size is known at the point where the loop occurs can be traversed this way:

```
extern double data[];  // array of unknown size

void f2()
{
    for (double& d : data)  // Error, data is an incomplete type.
    {
        // ...
    }
}

double data[10] = { /* ... */ };  // too late to make the above compile
```

The above example would compile if `data` were declared having a size, e.g., `extern double data[10]`, as that would be a complete type and provide sufficient information to traverse the array. The second definition of `data` in the example *is* complete but is not visible at the point that the loop is compiled.

An **initializer list** is typically used to initialize an array or container using **braced initialization**; see Section 2.1."??" on page ??. The `initializer_list` template does, however, provide its own `begin` and `end` member functions and is, therefore, directly usable as the *range-expression* in a ranged-based **for** loop:

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

void f3()
{
    for (double v : {1.9, 2.8, 4.7, 7.6, 11.5, 16.4, 22.3, 29.2, 37.1, 46.0})
    {
        // ...
    }
}
```

The example above shows how a series of **double** values can be embedded right within the loop header.

use-cases

Use Cases

elements-of-a-container

Iterating over all elements of a container

The motivating use case for this feature is to loop over the elements in a container:

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

void process(int* p);

void f1()
{
    std::list<int> aList{ 1, 2, 4, 7, 11, 16, 22, 29, 37, 46 };

    for (int& i : aList)
    {
        process(&i);
    }
}
```

This idiom takes advantage of all STL-compliant container types providing `begin` and `end` operations, which may be used to delimit a range encompassing the entire container. Thus, the loop above iterates from `aList.begin()` to `aList.end()`, calling `process` on each element encountered.

When iterating over a `std::map<Key, Value>` or `std::unordered_map<Key, Value>`, each element has type `std::pair<const Key, Value>`. To save typing and to avoid errors related to the first member of the pair being **const** we use the `value_type` alias to refer to each element's type:

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

using MapType = std::map<std::string, int>;
```

```
MapType studentScores
{
    {"Emily", 89},
    {"Joel",  85},
    {"Bud",   86},
};

void printScores()
{
    for (MapType::value_type& studentScore : studentScores)
    {
        const std::string& student = studentScore.first;
        int&               score   = studentScore.second;
        std::cout << student << "\t scored " << score << '\n';
    }
}
```

This example prints each key/value pair in the `map`. We create two aliases, `student` for `studentScore.first` and `score` for `studentScore.second`, to better express the intent of the code.[4]

Subranges

subranges

Using a classic **`for`** loop to traverse a container, `c`, allows a subrange of `c` to be specified beginning at some point after `c.begin()` — e.g., `++c.begin()` — and/or ending at some point before `c.end()` — e.g., `std::next(c.end(), -3)`. To specify a subrange for a ranged-based **`for`** loop, we create a simple adapter to hold two iterators (or iterator-like objects) that define the desired subrange:

```
template <typename Iter>
class Subrange
{
    Iter d_begin, d_end;

public:
    using iterator = Iter;

    Subrange(Iter b, Iter e) : d_begin(b), d_end(e) { }

    iterator begin() const { return d_begin; }
    iterator end()   const { return d_end;   }
};

template <typename Iter>
Subrange<Iter> makeSubrange(Iter beg, Iter end) { return {beg, end}; }
```

[4]In C++17, **structured bindings** allow two variables to be initialized from a single `pair`, each variable being initialized by the respective `first` and `second` members of the `pair`. A ranged-based **`for`** loop using a structured binding for the loop variables yields a very clean and expressive way to traverse containers like `map` and `unordered_map`, e.g., using `for (auto& [student, score] : studentScores)`.

The `Subrange` class above is a primitive start to a potentially rich library of range-based utilities.[5] It holds two externally-supplied iterators that it can supply to a ranged-based **for** loop via its `begin` and `end` accessor members. The `makeSubrange` factory uses function template argument deduction to return a `Subrange` of the correct type.[6]

Let's use `Subrange` to traverse a vector in reverse, omitting its first element:

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

template <typename Range>
void printRange(const Range& r)
{
    for (const auto& elem : r)
    {
        std::cout << elem << ' ';
    }

    std::cout << std::endl;
}

std::vector<int> vec{16, 3, 1, 8, 99};

void f1()
{
    printRange(makeSubrange(vec.rbegin(), vec.rend() - 1));
        // print "99 8 1 3"
}
```

The `printRange` function template will print out the elements of any range, provided the element type supports printing to a `std::ostream`. In `f1`, we use reverse iterators to create a `Subrange` starting from the last element of `vec` and iterating backwards. By subtracting `1` from `vec.rend()`, we exclude the last element of the sequence, which is the first element of `vec`.

In fact, the iterators need not refer to a container at all. For example, we can use `std::istream_iterator` to iterate over "elements" in an input stream:

```
#include <istream>   // std::istream
#include <iterator>  // std::istream_iterator
#include <sstream>   // std::istringstream

void f2()
{
    std::istringstream inStream("1 2 4 7 11 16 22 29 37 bad 46");
    printRange(makeSubrange(std::istream_iterator<int>(inStream),
                            std::istream_iterator<int>()));
}
```

[5]The C++20 Standard introduces a new Ranges Library that provides powerful features for defining, combining, filtering, and manipulating ranges.

[6]C++17 introduced **class template argument deduction**, which significantly reduces the need for factory templates like `makeSubrange`.

In `f2`, the range being printed uses the `istream_iterator<T>` adapter template. Each time through the loop, the adapter reads another `T` item from its input stream. At end-of-file or if a read error occurs, the iterator becomes equal to the sentinel iterator, `istream_iterator<T>()`. Note that the ranged-based **`for`** loop feature and the `Subrange` class template do not require that the size of the subrange be known in advance.

Range generators

range-generators

Iterating over a range does not necessarily entail traversing existing data elements. A range expression could yield a type that *generates* elements as it goes. A useful example is the `ValueGenerator`, an iterator-like class that produces a sequence of sequential values[7]:

```
template <typename T>
class ValueGenerator
{
    T d_value;

  public:
    explicit ValueGenerator(const T& v) : d_value(v) { }

    T operator*() const { return d_value; }
    ValueGenerator& operator++() { ++d_value; return *this; }

    friend bool operator!=(const ValueGenerator& a, const ValueGenerator& b)
    {
        return a.d_value != b.d_value;
    }
};

template <typename T>
Subrange<ValueGenerator<T>> valueRange(const T& b, const T& e)
{
    return { ValueGenerator<T>(b), ValueGenerator<T>(e) };
}
```

Instead of referring to an element within a container, `ValueGenerator` is an iterator-like type that *generates* the value returned by **`operator`**`*`. `ValueGenerator` can be instantiated for any type that can be incremented, e.g., integral types, pointers, or iterators. The `valueRange` function template is a simple factory to create a range comprising two `ValueGenerator` objects, using the `Subrange` class template defined in the use case described in Subranges. Thus, to print the numbers from 1 to 10, simply use a ranged-based **`for`** loop, employing a call to `valueRange` as the range expression:

```
void f3()
{
    // prints "1 2 3 4 5 6 7 8 9 10 "
    for (unsigned i : valueRange(1, 11))
```

[7]The `iota_view` and `iota` entities from the Ranges Library in the C++20 Standard provide a more sophisticated version of the `ValueGenerator` and `valueRange` facility described here.

```
    {
        std::cout << i << ' ';
    }
    std::cout << std::endl;
}
```

Note that the second argument to `valueRange` is one *past* the last item we want to iterate on, i.e., `11` instead of `10`. With something like `ValueGenerator` as part of a reusable utility library, this formulation expresses the intent of the loop more cleanly and concisely than the classic **`for`** loop.

The ability to generate numbers means that a range need not be finite. For example, we might want to generate a sequence of random numbers of indefinite length:

```
#include <random>  // std::default_random_engine, std::uniform_int_distribution

template <typename T = int>
class RandomIntSequence
{
    std::default_random_engine       d_generator;
    std::uniform_int_distribution<T> d_uniformDist;

public:
    class iterator
    {
        RandomIntSequence* d_sequence;

        explicit iterator(RandomIntSequence* s) : d_sequence(s) { }
        friend class RandomIntSequence;

    public:
        iterator& operator++() { return *this; }
        T operator*() const { return d_sequence->next(); }

        friend bool operator!=(iterator, iterator) { return true; }
    };

    RandomIntSequence(T min, T max, unsigned seed = 0)
        : d_generator(seed ? seed : std::random_device()())
        , d_uniformDist(min, max) { }

    T next() { return d_uniformDist(d_generator); }

    iterator begin() { return iterator(this); }
    iterator end()   { return iterator(nullptr); }
};

template <typename T>
RandomIntSequence<T> randomIntSequence(T min, T max, unsigned seed = 0)
{
    return {min, max, seed};
```

```
}
```

The `RandomIntSequence` class template uses the C++11 random-number library to generate high-quality pseudorandom numbers.[8] Each call to its `next` member function produces a new random number of integral type, `T`, within the inclusive range specified to the `RandomIntSequence` constructor. The nested `iterator` type holds a pointer to a `RandomIntSequence` and simply calls `next` each time it is dereferenced (i.e., via a call to **`operator`**`*`).

Of particular interest is **`operator`**`!=`, which returns **`true`** when comparing any two `RandomIntSequence<T>::iterator` objects. Thus, any ranged-based **`for`** loop that iterates over a `RandomIntSequence` is an infinite loop unless it terminates by some other means:

```
void f4()
{
    for (int rand : randomIntSequence(1, 10))
    {
        std::cout << rand << ' ';
        if (rand == 10) { break; }
    }

    std::cout << std::endl;
}
```

This example prints a list of random numbers in the range 1 through 10, inclusive. The loop terminates after printing `10` for the first (and only) time.

Iterating over simple values

ng-over-simple-values

The ability to iterate over a `std::initializer_list` can be very useful for processing a list of simple values or simple objects without first storing them in a container. Such a use case arises frequently when testing:

```
#include <limits>            // std::numeric_limits
#include <initializer_list>  // std::initializer_list

#define TEST_ASSERT(expr)  // ... assert that expr is true.

bool isEven(int i)
{
    return i % 2 == 0;
}

void testIsEven()
{
    // ...

    const int minInt = std::numeric_limits<int>::min();
    const int maxInt = std::numeric_limits<int>::max();
```

[8]An introduction to the C++11 random-number library can be found in Stephan T. Lavavej's excellent talk, `rand()` Considered Harmful.

```
    for (int testValue : {minInt, -256, -2, 0, 2, 4, maxInt - 1})
    {
        TEST_ASSERT(isEven(testValue));
        TEST_ASSERT(!isEven(testValue + 1));
    }
}
```

The `testIsEven` function iterates over a sample of numbers within the domain of `isEven`, including boundary conditions, testing that each number is correctly reported as being even and that adding 1 to the number produces a result that is correctly reported as not being even.

Initializer lists are not limited to primitive types, so the test data set can contain more complex values:

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

#define TEST_ASSERT_EQ(expr1, expr2)  // ... assert that expr1 == expr2.

int half(int i)
{
    return i / 2;
}

struct TestCase
{
    int value;
    int expected;
};

void testHalf()
{
    for (TestCase test : std::initializer_list<TestCase>{
        {-2, -1}, {-1, 0}, {0, 0}, {1, 0}, {2, 1}
    })
    {
        TEST_ASSERT_EQ(test.expected, half(test.value));
    }
}
```

In this case, the ranged-based **`for`** loop iterates over a `std::initializer_list` holding `TestCase` structures. This paring of input(s) with expected output(s) of a component under test is very common in unit tests.

potential-pitfalls

Potential Pitfalls

in-the-range-expression

Lifetime of temporary objects in the range expression

As described in *Description* on page 4 section, if the range expression evaluates to a temporary object, that object remains valid, as a result of lifetime extension, for the duration of the ranged-based **for** loop. Unfortunately, there are some subtle ways in which lifetime extension is not always sufficient.

The basic notion of lifetime extension is that, when bound to a reference, the lifetime of a *prvalue* — i.e., an object created by a literal, constructed in place, or returned (by value) from a function — is *extended* to match the lifetime of the reference to which it is bound:

```
#include <string>

std::string strFromInt(int);

void f1()
{
    const std::string& s1 = std::string('a', 2);
    std::string&&      s2 = strFromInt(9);
    auto&&             i  = 5;

    // s1, s2, and i are "live" here.

    // ...

}  // s1, s2, and i are destroyed at end of enclosing block.
```

The first string is constructed in place. The resulting temporary string would normally be destroyed as soon as the expression was complete but, because it is bound to a reference, its lifetime is extended; its destructor is not called and its memory footprint is not reused until `s1` goes out of scope, i.e., at the end of the enclosing block. The `strFromInt` function returns by value; the result of calling it in the second statement produces a temporary variable whose lifetime is similarly extended until `s2` goes out of scope. Finally, the forwarding reference, `i`, ensures that space in the current frame is allocated to hold the temporary copy of the (deduced) **`int`** value, `5`; such space cannot be reused until `i` goes out of scope at the end of the enclosing block. (See Section 2.1."**??**" on page **??**).

When the range expression for a range-based **for** loop is a *prvalue*, lifetime extension is vital to keeping the range object live for the duration of the loop:

```
void f2(int i)
{
    for (char c : strFromInt(i))
    {
        // ...
    }
}
```

The return value from `strFromInt` is stored in a temporary variable of type `std::string`. The temporary string is destroyed when the loop completes, not when the expression evaluation completes. If the string were to go out of scope immediately, it would not be possible to iterate over its characters. This code would have **undefined behavior** were it not for the lifetime extension harnessed by the ranged-based **for** loop.

The limitation of lifetime extension is that it applies only if the reference is bound *directly* to the temporary variable itself or to a subobject (e.g., a member variable) of the temporary variable, in which case the lifetime of the entire temporary variable is extended. Note that initializing a reference from a reference or a pointer to either the temporary or one of its subobjects does not count as binding *directly* to the temporary variable and does not trigger lifetime extension. The danger of an object getting destroyed prematurely is generally seen when the **full expression** returns a reference, pointer, or iterator into a temporary object:

```
#include <vector>   // std::vector
#include <string>   // std::string
#include <utility>  // std::pair
#include <tuple>    // std::tuple

struct Point
{
    double x, y;
    Point(double ax, double ay) : x(ax), y(ay) { }
};

struct SRef
{
    const std::string& str;
    SRef(const std::string& s) : str(s) { }
};

std::vector<int> getValues();  // Return a vector by value.

void f3()
{
    const Point& p1 = Point(1.2, 3.4);    // OK, extend Point lifetime.
    double&&     d1 = Point(1.2, 3.4).x;  // OK, extend Point lifetime.
    double&      d2 = Point(1.2, 3.4).y;  // Error, non-const lvalue ref, d2

    using ICTuple = std::tuple<int, char>;
    const int&  i1 = getValues()[0];                // Bug, dangling reference
    const int&  i2 = std::get<0>(ICTuple{0,'a'});  // Bug,      "         "
    auto&&      i3 = getValues().begin();           // Bug,      "    iterator
    const auto& s1 = std::string("abc").c_str();   // Bug,      "    pointer
    const auto& i4 = std::string("abc").length();  // OK, std::size_t extended

    SRef&&      sr = SRef("hello");  // Bug, string lifetime is not extended.
    std::string s2 = sr.str;         // Bug, string has been destroyed.
}
```

The first invocation of the `Point` constructor creates a temporary object that is bound to reference `p1`. The lifetime of this temporary object is extended to match the lifetime of the reference. Similarly, the lifetime of the second `Point` object is extended because a subobject, `x`, is bound to reference `d1`. Note that it is not permitted to bind a temporary to a non**`const`** *lvalue* reference, as is being attempted with `d2`, above.

The next four definitions do not result in useful lifetime extension at all.

1. In the case of `i1`, `getValues()` returns a *prvalue* of type `std::vector<int>`, resulting

in the creation of a temporary variable. That temporary variable, however, is *not* the value being bound to the `i1` reference; rather, the reference is bound to the result of the array-access operator (`operator[]`), which returns a reference into the temporary `vector` returned by `getValues()`. While we might consider an element of a `vector` logically to be a subobject of the vector, `i1` is not bound directly to that subobject but rather to the reference returned by `operator[]`. The vector goes out of scope immediately at the end of the statement, leaving `i1` to refer to an element of a deleted object.

2. The identical situation occurs with `i2` when accessing the member of a temporary `std::tuple`, this time via the nonmember function `std::get<0>`.

3. Rather than a reference, `i3` is deduced to be an *iterator* as the result of the expression. The iterator's lifetime is extended, but the lifetime of the object to which it refers is not.

4. Similarly, for `s`, the expression `std::string("abc").c_str()` yields a pointer into a temporary C-style string. Once again, the temporary `std::string` variable is not the object that is bound to the reference `s1`, so it gets destroyed at the end of the statement, invalidating the pointer.

Conversely, `i4` binds directly to the temporary object returned by `length`, extending its life even though the string itself gets destroyed as before. Unlike `i3` and `s1`, however, `i4` is not an iterator or pointer and so does not retain an implicit reference to the defunct string object.

The last two definitions, for `sr` and `s2`, show how subtle the rules for lifetime extension can be. The `"hello"` literal is converted into a temporary variable of type `std::string` and passed to the constructor of `SRef`, which *also* creates a temporary object. It is only the `SRef` object that is bound to the `sr` reference, so it is only the `SRef` object whose lifetime is extended. The `std::string("hello")` temporary variable gets destroyed when the constructor finishes executing, leaving the object referenced by `sr` with a member, `str`, that refers to a destroyed object.

There are good reasons why lifetime extension applies only to the temporary object being bound to a reference. A lot of code depends on temporary objects going out of scope immediately, i.e., to release a lock, memory, or some other resource. For range-based **for** loops, however, a compelling argument has been made that the correct behavior would be to extend the lifetime of *all* of the temporaries constructed while evaluating the range expression.[9] Unless and until this behavior is changed in a future Standard, beware of using a range expression that returns a reference to a temporary variable:

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

class RecordList
{
```

[9]At the time of writing the P2012 paper seeks to solve the issue when a range expression is a reference into a temporary. See *Fix the range-based for loop*, by Nicolai Josuttis, Victor Zverovich, Filipe Mulonde, and Arthur O'Dwyer, which references an original paper *Embracing Modern C++ Safely* by Rostislav Khlebnikov and John Lakos.

```
    std::vector<std::string> d_names;
    // ...

public:
    explicit RecordList(std::istream& is);
        // Create a RecordList with data read from is.

    // ...

    const std::vector<std::string>& names() const { return d_names; }
};

void printNames(std::istream& is)
{
    // Bug, RecordList's lifetime is not extended.
    for (const std::string& name : RecordList(is).names())
    {
        std::cout << name << '\n';
    }
}
```

The `RecordList` constructed in the range expression is not bound to the implied `__range` reference within the ranged-based **`for`** loop, so its lifetime will end before the loop actually begins. Thus, the **`const`** `std::vector<std::string>&` returned by its `names` method becomes a dangling reference, leading to undefined behavior (such as a segmentation fault).

To avoid this pitfall, create a named object for each temporary that you need to preserve, unless that temporary is the full expression for the range expression:

```
void printNames2(std::istream& is)
{
    {
        RecordList records(is);  // named variable
        for (const std::string& name : records.names())
        {
            std::cout << name << '\n';
        }

        // safe for records to go out of scope now
    }

    // ...
}
```

This minor rewrite of `printNames` creates an extra block scope in which we declare `records` as a named variable. The inner scope ensures that `records` gets destroyed immediately after the loop terminates.

t-copying-of-elements

Inadvertent copying of elements

When iterating through a container with a classic **for** loop, elements are typically referred to through an iterator:

std::vectorstd::string

```
void process(std::string&);

void f1(std::vector<std::string>& vec)
{
    for (std::vector<std::string>::iterator i = vec.begin();
         i != vec.end(); ++i)
    {
        process(*i);  // refer to element via iterator
    }
}
```

The ranged-based **for** loop gives the element a name and a type. If the type is not a reference, then each iteration of the loop will *copy* the current element. In many cases, this copy is inadvertent:

```
void f2(std::vector<std::string>& vec)
{
    for (std::string s : vec)
    {
        process(s);  // call process on *copy* of string element, potential
                     // bug.
    }
}
```

The example above illustrates two issues: (1) there is an unnecessary expense in copying each string, and (2) `process` may modify or take the address of its argument, in which case it will modify or take the address of the copy, rather than the original element; the strings in `vec` will remain unchanged.

This error appears to be especially common when using **auto** to deduce the loop variable's type:

```
void f3(std::vector<std::string>& vec)
{
    for (auto s : vec)
    {
        process(s);  // call process on *copy* of deduced string element,
                     // potential bug.
    }
}
```

Copying an element is not always erroneous, but it may be wise to habitually declare the loop variable as a reference, making deliberate exceptions when needed:

```
void f4(std::vector<std::string>& vec)
{
    for (std::string& s : vec)
```

```
    {
        process(s);  // OK, call process on *reference* to string element
    }
}
```

If we wish to avoid copying elements but also wish to avoid modifying them, then a **`const`** reference will provide a good balance. Note, however, that if the type being iterated over is not the same as the type of the reference, a conversion might quietly produce the (undesired) copy anyway:

```
void f5(std::vector<char*>& vec)
{
    for (const std::string& s : vec)
    {
        // s is a reference to a *copy* of an element of vec.
    }
}
```

In this example, the elements of `vec` have type **`char`**`*`. The use of **`const`** `std::string&` to declare the loop variable `s` correctly prevents modification of any elements of `vec`, but there is still a copy being made because each member access is converted to an object of type `std::string`.

For generic code that modifies a container, **`auto`**`&&` is the most general way to declare the loop variable. For generic code that does not modify the container, **`const auto`**`&` is safer:

```
template <typename Rng>
void f6(Rng& r)
{
    for (auto&& e : r)
    {
        // ...
    }

    for (const auto& cr : r)
    {
        // ...
    }
}
```

Because `e` is a forwarding reference and `cr` is a **`const`** reference, they will both correctly bind to the return type of `*begin(Rng)`, even if that type is a *prvalue*. Note that the use of **`auto`** can obfuscate code by hiding the underlying types of objects. Obfuscated code is prone to bugs so these uses of **`auto`** are recommended chiefly for *generic* code or where other trade-offs favor using this short-cut; see Section 2.1."**??**" on page **??**.

Simple and reference-proxy behaviors can be be different

rs-can-be-be-different

Some containers have iterators that return proxies rather than references to their elements. Depending on how the loop variable is declared, the unwary programmer may get surprising results when the container's iterator type returns reference proxies.

An example of such a container is `std::vector<bool>`. The type, `std::vector<bool>::iterator::reference` is a reference-proxy class that emulates a reference to a single bit within the `vector`. The proxy class provides an **`operator bool()`** that returns the bit when the proxy is converted to **`bool`** and an **`operator=(bool)`** that modifies the bit when assigned a Boolean value.

Let's consider a set of loops, each of which iterates over a `vector` and attempts to set each element of the vector to **`true`**. We'll embed the loops in a function template so that we can compare the behavior of instantiating with a normal container (`std::vector<int>`) and with one whose iterator uses a reference proxy (`std::vector<bool>`):

```
#include <vector>

template <typename T>
void f1(std::vector<T>& vec)
{
    for (T      v : vec) { v = true; }  // (1)
    for (T&     v : vec) { v = true; }  // (2)
    for (T&&    v : vec) { v = true; }  // (3)
    for (auto   v : vec) { v = true; }  // (4)
    for (auto&  v : vec) { v = true; }  // (5)
    for (auto&& v : vec) { v = true; }  // (6)
}

void f2()
{
    using IntVec  = std::vector<int>;   // has normal iterator
    using BoolVec = std::vector<bool>;  // has iterator with reference proxy

    IntVec  iv{ /* ... */ };
    BoolVec bv{ /* ... */ };

    f1(iv);
    f1(bv);
}
```

For each of the loops in `f1`, the difference in behavior between the `IntVec` and `BoolVec` instantiations hinges on what happens when `v` is initialized from `*__begin` within the loop transformation. For the `IntVec` iterator, `*__begin` returns a *reference* to the element within the container. For the `BoolVec` iterator, it returns an *object* of the reference-proxy type.

- *Loop (1)* produces identical behavior from both instantiations. The loop makes a local copy of each element, and then modifies the copy. The only difference is that the `BoolVec` version performs a conversion to **`bool`** to initialize `v`, whereas the `IntVec` version initializes `v` directly from the element reference. For both the `IntVec` or `BoolVec` version, the fact that the original vector is unchanged is a potential bug (see Inadvertent copying of elements, above).

- *Loop (2)* modifies the container elements in the `IntVec` instantiation but fails to compile for the `BoolVec` instantiation. The compilation error comes from trying to initialize the non**const** *lvalue* **reference**, v, from an *rvalue* of the proxy type. The **bool** conversion operator does not help since the result would still be an *rvalue*.

- *Loop (3)* fails to compile for the IntVec iterator because the ***rvalue* reference**, v, cannot be initialized from *__begin, which is an *lvalue* reference. Surprisingly, the BoolVec instantiation *does* compile, but the loop does not modify the container. Here, **operator bool** is invoked on the proxy object returned by *__begin. The resulting temporary object is bound to v, and its lifetime is extended for the duration of the iteration. Because v is bound to a temporary variable, modifying v modifies only the temporary, not the original, element, resulting in a likely bug as in the case of loop (1).

- *Loop (4)* compiles for both the BoolVec and IntVec cases but produces different results for each. For IntVec iterators, **auto** deduces the type of v as **int**, so assigning to v modifies a local copy of the element, as in loop (1). For BoolVec iterators, the deduced type of v is the proxy type rather than **bool**. Assigning to the proxy *does* change the element of the container.

- *Loop (5)*, like loop (2), works as expected for IntVec instantiation but fails to compile for the BoolVec instantiation. As before, the problem is that the BoolVec iterator yields an *rvalue* that cannot be used to initialize an *lvalue* reference.

- *Loop (6)* produced identical behavior from both instantiations, modifying each of the vector elements. The type of v is deduced to be **int**& for IntVec instantiation and the proxy type for the BoolVec instantiation. Assigning through either the real reference or the reference proxy modifies the element in the container.

Let's now look at the the situation with **const**-qualified loop variables:

```
template <typename T>
void f3(std::vector<T>& vec)
{
    for (const T      v : vec) { /* ... */ }  // (7)
    for (const T&     v : vec) { /* ... */ }  // (8)
    for (const T&&    v : vec) { /* ... */ }  // (9)
    for (const auto   v : vec) { /* ... */ }  // (10)
    for (const auto&  v : vec) { /* ... */ }  // (11)
    for (const auto&& v : vec) { /* ... */ }  // (12)
}

void f4()
{
    using IntVec  = std::vector<int>;   // has normal iterator
    using BoolVec = std::vector<bool>;  // has iterator with reference proxy

    IntVec  iv{ /* ... */ };
    BoolVec bv{ /* ... */ };

    f3(iv);
```

```
    f3(bv);
}
```

- *Loop (7)* works identically for both instantiations, converting the proxy reference to **bool** in the BoolVec case.
- *Loop (8)* works identically for both instantiations. For the IntVec case, the result of `*__begin` is bound directly to v. For the BoolVec case, `*__begin` produces proxy reference that is converted to a **bool** temporary that is then bound to v. Lifetime extension keeps the bool value alive.
- *Loop (9)* fails to compile for IntVec but succeeds for BoolVec exactly as for loop (3) except that the temporary **bool** bound to v is **const** and thus does not risk giving programmers the false belief that they are modifying the container.
- *Loop (10)* has the same behavior for both the IntVec and BoolVec instantiations. That mechanism is the same behavior as for loop (4) except that, because v is **const**, *neither* instantiation can modify the container.
- *Loop (11)* also works for both instantiations. For the IntVec case, the result of `*__begin` is bound directly to v. For the BoolVec case, v is deduced to be a **const** reference to the proxy type; `*__begin` produces a temporary variable of the proxy type, which is then bound to v. Lifetime extension keeps the proxy alive. In most contexts, a **const** proxy reference is an effective stand-in for a **const bool**&.
- *Loop (12)* fails to compile for IntVec but succeeds for BoolVec. The error with IntVec occurs because **const auto**&& is always a **const** *rvalue* reference (not a forwarding reference) and cannot be bound to the *lvalue* reference, `*__begin`. For BoolVec, the mechanism is identical to loop (11) except that loop (11) binds the temporary object to an *lvalue* reference whereas loop (12) uses an *rvalue* reference. When the references are **const**, however, there is little practical differences between them.

Note that loops 4, 6, 10, 11, and 12 in the BoolVec instantiations bind a reference to a temporary proxy reference object, so taking the address of v in these situations is likely not to produce useful results. Additionally, loops 3, 8, and 9 bind v to a temporary **bool**. Users must be mindful of the lifetime of these temporary objects (a single-loop iteration) and not allow the address of v to escape the loop.
Proxy objects emulating references to non-class elements within a container are surprisingly effective, but their limitations are exposed when they are bound to references. In generic code, as a rule of thumb, **const auto**& is the safest way to declare a read-only loop variable if a reference proxy might be in use, while **auto**&& will give the most consistent results for a loop that modifies its container. Similar issues, unrelated to range-based **for** loops, occur when passing a proxy reference to a function taking a reference argument.

Annoyances

annoyances

No access to the state of the iteration

tate-of-the-iteration

When traversing a range with a classic **for** loop, the loop variable is typically an iterator or array index. Within the loop, we can modify that variable to repeat or skip iterations.

Similarly, the loop-termination condition is usually accessible so that it is possible to, for example, insert or remove elements and then recompute the condition:

```
#include <string>  // std::string
#include <cctype>  // std::isupper

void spaceBeforeCaps(std::string& s)
{
    // Insert a space before each capital letter in s.
    using IdxType = std::string::size_type;
    for (IdxType i = 0; i < s.size(); ++i)
    {
        if (std::isupper(s[i]))
        {
            s.insert(i, 1, ' ');  // Insert one space at i.
            ++i;                  // Skip the space.
        }
    }
}
```

The code above depends on (1) having access to the iteration index, (2) being able to change the iteration index, and (3) recomputing the size of the collection each time through the loop. No similar function could be written using a ranged-based **for** loop since the `__range`, `__begin`, and `__end` variables are for exposition only and are not accessible from within the code:

```
void spaceBeforeCaps2(std::string& s)
{
    // Insert a space before each capital letter in s.
    for (char c : s)
    {
        if (std::isupper(c))
        {
            __begin = s.insert(__begin, ' ');  // Error, no __begin variable
            ++__begin;                         // Error, no __begin variable
            __end = s.end();                   // Error, no __end variable
        }
    }
}
```

A classic **for** loop can traverse more than one container at a time (e.g., to add corresponding elements from two containers and store them into a third). It accomplishes this feat by either incrementing multiple iterators on each iteration or keeping a single index that is used to access multiple, random-access iterators concurrently. Trying to accomplish something similar with a range-based **for** loop usually involves using a hybrid approach:

```
#include <vector>   // std::vector
#include <cassert>  // standard C assert macro

void addVectors(std::vector<int>&       result,
                const std::vector<int>& a,
```

```
                 const std::vector<int>& b)
    // For each element ea of a and corresponding element eb of b, set
    // the corresponding element of result to ea + eb.  The behavior is
    // undefined unless a and b have the same length.
{
    assert(a.size() == b.size());
    result.resize(a.size());

    std::vector<int>::const_iterator ia = a.begin();
    std::vector<int>::const_iterator ib = b.begin();
    for (int& sum : result)
    {
        sum = *ia++ + *ib++;
    }
}
```

Although `result` is traversed using the range-based **for** loop, `a` and `b` are effectively traversed manually using iterators. It is arguable as to whether the code is any clearer or simpler to write than it would have been using a classic **for** loop.

This situation can be improved through the use of a "zip" iterator — a type that holds multiple iterators and increments them in lock-step. Using a "zip" iterator, all three containers can be traversed using a single ranged-based **for** loop:

```
#include <cassert>  // standard C assert macro
#include <tuple>    // std::tuple
#include <utility>  // std::declval
#include <vector>   // std::vector

template <typename... Iter>
class ZipIterator
{
    std::tuple<Iter...> d_iters;

    // ...

public:
    using reference = std::tuple<decltype(*std::declval<Iter>())...>;

    ZipIterator(const Iter&... i);

    reference operator*() const;
    ZipIterator& operator++();
    friend bool operator!=(const ZipIterator& a, const ZipIterator& b);
};

template <typename... Range>
class ZipRange
{
    using ZipIter =
        ZipIterator<decltype(begin(std::declval<Range>()))...>;
```

```
    // ...

  public:
    ZipRange(const Range&... ranges);

    ZipIter begin() const;
    ZipIter end() const;
};

template <typename... Range>
ZipRange<Range...> makeZipRange(Range&&... r);

void addVectors2(std::vector<int>&       result,
                 const std::vector<int>& a,
                 const std::vector<int>& b)
{
    assert(a.size() == b.size());
    result.resize(a.size());

    for (std::tuple<int, int, int&> elems : makeZipRange(a, b, result))
    {
        std::get<2>(elems) = std::get<0>(elems) + std::get<1>(elems);
    }
}
```

Each iteration, instead of yielding a single element, yields a `std::tuple` of elements resulting from the traversal of multiple ranges simultaneously. To be used, the elements must be unpacked from the `std::tuple` using `std::get`. Zip iterators become much more attractive in C++17 with the advent of structured bindings, which allow multiple loop variables to be declared at once, without the need to directly unpack the `std::tuple`s. The above implementation and usage of `ZipRange` is just a rough sketch; the full design and implementation of zip iterators and zip ranges is beyond the scope of this section.

Adapters are required for many tasks

required-for-many-tasks

In the usage examples above, we have seen a number of adapters (e.g., to traverse subranges, to traverse a container in reverse, to generate sequential values), and zip iterators to iterate over multiple ranges at once.

None of these adapters would be required for a classic **for** loop. On the one hand, one-off situations are expressed more simply with a classic **for** loop. On the other hand, the adapters that we create to make range-based **for** loops usable in more situations can lead to the development of a reusable *library* of adapters. Using the `ValueGenerator` class from Range generators, for example, produces simpler and more expressive code than using a classic **for** loop would.[10]

[10] The Standard's Ranges Library, introduced in C++20, provides a sophisticated algebra for working with and adapting ranges.

ntinel-iterator-types

No support for sentinel iterator types

For a given range expression, `__range`, `begin(__range)`, and `end(__range)` must return the same type to be usable with a ranged-based **`for`** loop. This limitation is problematic for ranges of indeterminate length, where the condition for ending a loop is not determined by comparing two iterators. For example, in the `RandomIntSequence` example (see Range generators), the end iterator for the infinite random sequence holds a null pointer and is never used, not even within **`operator`**`!=`. It would be more efficient and convenient if the end iterator were a special, empty *sentinel* type. Comparing any iterator to the sentinel would determine whether the loop should terminate:

```
template <typename T = int>
class RandomIntSequence2
{
    // ...

public:
    class sentinelIterator { };

    class iterator {
        // ...
        friend bool operator!=(iterator, sentinelIterator) { return true; }
    };

    iterator         begin() { /* ... */ }
    sentinelIterator end() const { return {}; }
};
```

The above code shows an example of `begin` and `end` returning different types, where `end` returns an empty sentinel type. Unfortunately, using this formulation of `RandomIntSequence2` with a C++11 range-based **for** loop will result in a compilation error complaining that `begin` and `end` return inconsistent types.[11]

Another type that could benefit from a sentinel iterator is `std::istream_iterator`, since the state of the end iterator is never used. It is unlikely that this interface will change, however, as `std::istream_iterator` has been with us since the first C++ Standard.

Only ADL lookup

only-adl-lookup

The free functions `begin` and `end` are found using argument-dependent lookup (ADL) only. File-scope (**`static`**) functions are not considered. If we wish to add `begin` and `end` functions for a range-like type that we do not own, we need to put those functions into the same namespace as the range-like type, inviting a potential name collision with other compilation units attempting to do the same thing:

```
// third_party_library.h:

namespace third_party
{

    class IteratorLike { /* ... */ };

    class RangeLike
    {
        // ... does not provide begin and end members
    };

    // ... does not provide begin and end free functions
}
```

[11] This limitation on the use of sentinel iterators was lifted in C++17. Sentinel iterators are supported directly by the C++20 Ranges Library. In C++17, the specification was modified to:

```
{
    auto&& __range = range-expression;
    auto __begin   = begin-expr;
    auto __end     = end-expr;
    for (; __begin != __end; ++__begin)
    {
        for-range-declaration = *__begin;
        statement
    }
}
```

```
// myclient.cpp:

#include <third_party_library.h>

static third_party::IteratorLike begin(third_party::RangeLike&);
static third_party::IteratorLike end(third_party::RangeLike&);

void f()
{
    third_party::RangeLike rl;
    for (auto&& e : rl)  // Error, begin not found by ADL
    {
        // ...
    }
}
```

The code above attempts to work around the absence of `begin(RangeLike&)` and `end(RangeLike&)` in the third-party library by defining them locally within `myclient.cpp`. This attempt fails because static functions are not found via ADL. A better workaround that does work is to create a range adapter for the range-like class:

```
class RangeLikeAdapter
{
    // ...

public:
    RangeLikeAdapter(third_party::RangeLike&);
    third_party::IteratorLike begin() { /* ... */ }
    third_party::IteratorLike end()   { /* ... */ }
};
```

The adapter wraps the range-like type and provides the missing features. Beware, however, that if the wrapper stores a pointer or reference to a temporary `RangeLike` object, you don't run into the pitfall where the lifetime of temporary objects is not always extended; see Lifetime of temporaries in the range expression.

`see-also`

See Also

- "**??**" (§2.1, p. **??**) ♦ explains **`auto`**, often used in a ranged-based **`for`** loop to determine the type of the loop variable, and many of the pitfalls of **`auto`** apply when using it for that purpose.
- "**??**" (§3.1, p. **??**) ♦ show how to overload member functions to work differently on *rvalues* and *lvalues*.

`further-reading`

Further Reading

No further reading.

sec-conditional-cpp14

Chapter 3

Unsafe Features

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

sec-unsafe-cpp14