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.

List Initialization: std::initializer_list<T>

::initializer_list initlist

The C++ Standard Library's `std::initializer_list` class template supports lightweight, compiler-generated arrays of values that are initialized in source code similarly to built-in, C-style arrays using the generalized braced initialization syntax.

Description

description

C++, and even C before it, allowed built-in arrays to be initialized via brace-enclosed lists of values:

```
int data[] = { 0, 1, 1, 2, 3, 5, 8, 13 };  // initializer list of 8 int values
```

C++11 extends this concept to allow such lists of values to be provided to **user-defined types (UDTs)** in a variety of circumstances. The compiler arranges for the values to be stored in an unnamed C-style array, and nonmodifiable access to that array is provided via the `std::initializer_list` class template. This template provides a programmer-accessible API and semantics inspired by a nonmodifiable, C++ standard container but having reference semantics rather than ownership of the underlying array. The C++ Standard provides a reference definition that comprises **`typedef`**s, accessors, and an explicitly declared default constructor, along with implicit definitions of the other five **special member functions**; see Section 1.1."**??**" on page **??**:

```
namespace std
{

template <typename E>
class initializer_list  // illustration of programmer-accessible interface
{
    const E* d_begin_p;  // pointer to the beginning of a contiguous array
    const E* d_end_p;    // pointer to one past the end of the array (or 0)
                         // could have used size_t as an offset instead.

    // ... (inaccessible, used by the compiler to initialize this object)
public:
    typedef E value_type;               // C++ type of each array element
    typedef const E& reference;         // There is no nonconst reference.
    typedef const E& const_reference;   // const lvalue reference type
    typedef size_t size_type;           // type returned by size()

    typedef const E* iterator;          // There is no nonconst iterator.
    typedef const E* const_iterator;    // const element-iterator type

    constexpr initializer_list() noexcept;  // default constructor

    constexpr size_t size() const noexcept;     // number of elements
    constexpr const E* begin() const noexcept;  // beginning iterator
    constexpr const E* end() const noexcept;    // one-past-the-last iterator
};
// initializer list range access
template <typename E> constexpr const E* begin(initializer_list<E> il) noexcept;
template <typename E> constexpr const E* end(initializer_list<E> il) noexcept;

}  // close std namespace
```

The reference implementation in the code example above illustrates the public functionality available for direct use by the compiler and programmers alike and elides the private machinery used by the compiler to initialize objects other than an empty initializer list. Objects of this template, instantiated for element type `E`, act as lightweight proxies for the compiler supplied arrays that, when copied or assigned, do not copy the underlining elements. Access to the elements is provided by the member and/or free functions `begin` and `end`, which satisfy the Standard Library requirements of a range with random access iterators.

The accessible interface of the `std::initializer_list` class template, in the code example above, also employs two other C++11 language features: **`constexpr`** and **`noexcept`**. The **`constexpr`** keyword allows the compiler to consider using a function so decorated as part of a **constant expression**; see Section 2.1."**??**" on page **??**. The **`noexcept`** specifier indicates that this function is not allowed to throw an exception; see Section 3.1."**??**" on page **??**.

As an introductory example, consider a function, `printMembers`, that will print the elements of a given sequence of integers that is represented by its `std::initializer_list<int>` parameter, `il`:

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

void printNumbers(std::initializer_list<int> il)  // prints given list of ints
{
    std::cout << "{";
    for (const int* ip = il.begin(); ip != il.end(); ++ip)  // classic for loop
    {
        std::cout << ' ' << *ip;  // output each element in given list of ints
    }
    std::cout << " } [size = " << il.size() << ']';
}
```

Using member functions `begin` and `end`, the `printNumbers` function in the code snippet above employs, for exposition purposes, the classic **`for`** loop to iterate through the supplied initializer list, printing each of the elements in turn to `stdout`, eventually followed by the size of the list. Note that `il` is passed by value rather than by **`const`** reference; this style of parameter passing is used purely as a matter of convention, because `std::initializer_list` is designed to be a small *trivial* type that many C++ implementations can optimize by efficiently passing such function arguments using CPU registers.

We can now write a `test` function to, in turn, invoke this `printNumbers` function on an initializer list denoted using braced-initialization syntax; see Section 2.1."**??**" on page **??**:

```
void test()
{
    printNumbers({ 1, 2, 3, 4 });  // prints "{ 1 2 3 4 } [size = 4]"
}
```

In the `test` function above, the compiler transforms the braced expression `{ 1, 2, 3, 4 }` into a **temporary** unnamed C-style array of nonmodifiable values of type **`int`**, which is then passed via an `std::initializer_list<int>` to `printNumber` for processing. While the `std::initializer_list` temporary does not own the temporary array, their lifetimes are the same, so there is no risk of a dangling reference.

Using the `initializer_list` class template

r_list-class-template

When a program uses a `std::initializer_list`, either by mentioning it explicitly or having it implicitly created by the compiler, the program must include — directly or indirectly — the `<initializer\_list>` header, or else the program is ill formed:

```
std::initializer_list<int> x = { };  // Error, <initializer_list> not included
void f(std::initializer_list<double>);  // Error,    "         "      "      "
auto ilx = {1,2,3};                     // Error,    "         "      "      "
auto ilv { 1 };  // OK, but be aware that direct list initialization deduces int

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

std::initializer_list<int> y = { };     // OK
void f(std::initializer_list<double>);  // OK
auto ily = {1,2,3};                     // OK
auto ilz = { };                         // Error, cannot deduce element type
auto ilw = { 1 };                       // OK, std::initializer_list<int>
```

In the example code above, any explicit or implicit use of the `std::initializer_list` requires the compiler to have first seen its definition — even when the type is deduced using **`auto`**; see Section 2.1."**??**" on page **??**. Note, in particular, the initialization of `ilv` omits the equal (`=`) symbol, forcing **`auto`** to deduce a copy of the argument rather than an `std::initializer_list`; see Section 2.1."**??**" on page **??**.

For a given element type, `E`, the `std::initializer_list<E>` class defines several type aliases, similar to how they are defined for standard-library containers:

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

struct E { };
std::initializer_list<E>::size_type       ts;        // ts  of type std::size_t
std::initializer_list<E>::value_type      tv;        // tv  of type E
std::initializer_list<E>::reference       tr  = tv;  // tr  of type const E&
std::initializer_list<E>::const_reference tcr = tv;  // tcr of type const E&
std::initializer_list<E>::iterator        ti;        // tr  of type const E*
std::initializer_list<E>::const_iterator  tci;       // tcr of type const E*
```

Note that both `reference` and `iterator` are **`const`**, just like `const_reference` and `const_iterator`.

Compiler vendors are permitted to and often do provide alternative types for such aliases in general purpose containers — e.g., "debugging" iterator to assist in detecting errors such as dereferencing a past-the-end iterator. In the case of `std::initializer_list`, however, these type aliases are fixed in the C++ Standard, so they cannot vary.

A public default constructor is provided, which enables clients to create an empty initializer list:

```
std::initializer_list<E> x;  // x is an empty list type E elements.
```

Each of the other special member functions, such as the copy constructor and copy assignment operator, is implicitly generated and available to public clients:

```
std::initializer_list<E> y(x);  // copy construction
void assignX2Y() { y = x; }     // copy assignment
```

Note that there is no public constructor for creating an `std::initializer_list` in any other way; the intention is that `std::initializer_list` objects are created behind the scenes — *by the compiler* — from braced lists:

```
E e0, e1, e2, e3;
std::initializer_list<E> z({e0, e1, e2, e3});   // copy construction (BAD IDEA)
void assignTemporary2y() { y = {e3, e2, e1}; }  // copy assignment (BAD IDEA)
```

Be warned, however, that although objects of type `std::initializer_list<E>` may be copied and assigned freely, doing so does not copy the underlying array. Importantly, creating a temporary initializer list and copy constructing (or copy assigning) its "value" to another does *not* extend the lifetime of its underlying array beyond the end of outermost containing *expression* in which it is used; hence, any subsequent attempt to access any of the *data* of any of the elements of that underlying array will result in **undefined behavior**; see **Pointer semantics** and lifetimes of temporaries.

The `size() const` member function returns the number of elements referenced by an initializer list:

```
int nX = x.size();  // OK, nX == 0 -- { }
int nY = y.size();  // OK, nY == 0 -- { }
int nZ = z.size();  // BAD IDEA, nZ == 4 -- Oops!
```

Note that calling `z.size()` may seem well defined even after the associated temporary array has expired. For implementations that choose a pointer and length representation, this would be true. However, for implementations choosing a two-pointer representation, the result of subtracting two invalid pointers (since the array no longer exists) is implementation-defined behavior and could, depending on the implementation's design, signal or trap to expose use of bad pointers. Accessing data in the underlying array is always undefined behavior at this point. Given that the choice of implementation is unspecified and left to the library vendor, it is best to avoid relying on any functions that query an `std::initializer_list` outside the lifetime of its underlying array, just as we would not dereference an invalid pointer.

Since `size`, like all of the member functions of an `std::initializer_list`, is declared **`constexpr`** (see Section 2.1."**??**" on page **??**), an `std::initializer_list` is considered a **literal type** and, hence, is eligible to be evaluated within a constant expression; see Section 1.1."**??**" on page **??**:

```
static_assert(std::initializer_list<int>({0, 1, 2, 3}).size() == 4, "");  // OK

int a[(std::initializer_list<int>({2, 1, 0}).size())];  // a is an array of 3 int.
```

Accessing the members of the underlying array of a `std::initializer_list<E>` is accomplished via iterators of type `E*`: two **`const`** (**`constexpr`**) member functions, `begin()` and `end()`, return pointers to, respectively, the first and the one-past-the-end array-element positions:

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

void test1()
{
    std::initializer_list<char> list({'A', 'B', 'C'});
    assert(*list.begin()   == 'A');                     // front element
    assert(list.begin()[0] == 'A');                     // element 0
    assert(list.begin()[1] == 'B');                     // element 1
    assert(list.begin()[2] == 'C');                     // element 2
    assert(list.begin() + 3           == list.end());  // true in this case
    assert(list.begin() + list.size() == list.end());  // always true
}
```

In the `test1` function in the code example above, `list.begin()` serves as the address of both the first element and that of the underlying contiguous array of elements itself — suitable for application of the built-in index operator (`[]`), although that is not how `begin()` is typically used. While `begin()` is a pointer to the first element of the range, `end()` is a pointer to one-past-last element and `size()` == `end()` - `begin()`. If `size()` == `0`, the value of `begin()` and `end()` are unspecified but must be equal.

The presence of the `begin()` and `end()` member functions enable an `std::initializer_list` to be used as the source of a range-based **`for`** loop; see Section 2.1.“**??**” on page **??**:

```
void test2()  // print "10 20 30 " to stdout
{
    std::initializer_list<int> il = { 10, 20, 30 };

    for (int i : il)
    {
        std::cout << i << ' ';
    }
}
```

Moreover, these member functions enable us to specify a **braced-initializer list** as the source range of a range-based **`for`** loop:

```
void test3()  // print "100 200 300 " to stdout
{
    for (int i : {100, 200, 300})
    {
        std::cout << i << ' ';
    }
}
```

Note that the use of a temporary `std::initializer-list`, as in the example above, is supported in a range-base **`for`** loop only because life-time extension (i.e., via binding to a reference as opposed to copying) of this library object is magically tied by the language to a corresponding lifetime extension of the underlying array. Without lifetime extension, this too would have been considered undefined behavior; again see *Pointer semantics and lifetimes of temporaries* on page 10.

Finally, corresponding, global `std::begin` and `std::end` **free-function** templates are overloaded for `std::initializer_list` objects directly in the `<initializer_list>` header — rather than relying on the more general templates in the `<iterator>` header — since `<initializer_list>` is part of the minimal, **freestanding** subset of the C++ Standard Library and `<iterator>` is not:

```
template <typename E>
constexpr const E* begin(std::initializer_list<E> list) noexcept
{
    return list.begin();  // invokes member function begin()
}

template <typename E>
constexpr const E* end(std::initializer_list<E> list) noexcept
{
    return list.end();  // invokes member function end()
}
```

As it turns out, these functions were originally added to support C++11's range-based **`for`** loop, but a subsequent design change rendered them superfluous; see *Annoyances — Overloaded free-function templates* `begin` *and* `end` *are largely vestigial* on page 22. Their only remaining use is to support locally a style of generic programming whereby free, rather than member, functions are strongly preferred:

```
#include <initializer_list>  // std::initializer_list
#include <algorithm>         // std::min, std::max

template <typename T>
T minVal(std::initializer_list<T> il)  // implemented with member functions
{
    return std::min(il.begin(), il.end());  // invokes member begin, end
}

template <typename T>
T maxVal(std::initializer_list<T> il)  // implemented using free functions
{
    return std::max(begin(il), end(il));  // invokes free begin, end
}
```

Without the locally overloaded free-function versions of `begin` and `end` in `<initializer_list>`, it would have been necessary to include the `<iterator>` header along with all that entails to get the implementation of `maxVal` above to compile.

Pointer semantics and lifetimes of temporaries

etimes-of-temporaries

An instance of `std::initializer_list` class template is a lightweight proxy for a homogeneous array of values. This type does not itself contain any data but instead refers to the data via the address of that data. It is unspecified whether a `std::initializer_list` is implemented as a pair of pointers or a pointer and a length.

Any nonempty array of values to which an `std::initializer_list` refers is created by the compiler in the program context in which the braced list of values appears. Each such compiler-generated array is — at least in principle — a temporary object having the same lifetime as other temporaries created in that context. The `std::initializer_list` object itself has a special form of *pointer semantics* understood by the compiler, such that the lifetime of the temporary array will be extended to the lifetime of the `std::initializer_list` object for which the underlying array was created. Importantly, the lifetime of this underlying array is *never* extended by copying its proxy initializer-list object.

Consider a `std::initializer<int>`, `il`, initialized with three numbers, `1`, `2`, and `3`:
std::initializer_list

```
std::initializer_list<int> il = {1, 2, 3};  // initializes il with 3 numbers
```

The compiler first creates a temporary array holding the three numbers. That array would normally be destroyed at the end of the outermost expression in which it appeared, but initializing `il` to refer to this array extends its lifetime to be coterminous with `il`. There are several other ways one might try to initialize such a list:

```
typedef std::initializer_list<int> Ili;
Ili il0 =  {1, 2, 3};   // OK, copy initialization (implicit ctors only)
Ili iL0ne  {1, 2, 3};   // OK, direct     "        (even explicit ctors)
Ili iL1 =  (1, 2, 3);   // Error, conversion from int to nonscalar requested
Ili iL1ne  (1, 2, 3);   // Error, no matching function call for (int, int,int)

Ili jL2 = ({1, 2, 3});  // Error, illegal context for statement expression
Ili jL2ne ({1, 2, 3});  // OK, direct initialization via a single init-list arg
Ili jL3 = ((1, 2, 3));  // Error, conversion from int to nonscalar requested
Ili jL3ne ((1, 2, 3));  // Error, no matching function call for (int)

Ili kL4 = {{1, 2, 3}};  // Error, conversion from brace-enclosed list requested
Ili kL4ne {{1, 2, 3}};  // Error,    "       "     "      "      "       "
Ili kL5 = {(1, 2, 3)};  // Bug, copy initialization to single-int int list
Ili kL5ne {(1, 2, 3)};  // Bug, direct       "      "    "      "     "
```

As can be inferred from the code example above, the language treats direct and copy initialization of a `std::initializer_list` the same — i.e., as if the inaccessible constructor used by the compiler to populate an `std::initializer_list` is declared *without* the **`explicit`** keyword; see Section 2.1."**??**" on page **??**.

Copying such a temporary to an *existing* initializer list, however, would be ill advised:

```
void assign3InitializerList()  // BAD IDEA
{
    il = { 4, 5, 6, 7 };  // il has dangling reference to a temporary array
}
```

The temporary array created in the assignment expression above is not used to initialize `il`, so that temporary array's lifetime is not extended; it will be destroyed at the end of the assignment expression, leaving `il` as a dangling reference to a list of numbers that no longer exists.

Some C++ implementations permit us to still access the number of elements that were assigned to `il`, though note that calling even `size` on an `std::initializer_list` bound to an expired array does not have predictable behavior across all C++ implementations; see *Using the* `initializer_list` *class template* on page 6:

```
int size = il.size();  // size == 4
```

But if we were to try to access any specific element in the array, that behavior would be undefined:

```
int firstValue = *il.begin();  // undefined behavior; accessing expired value
```

ializer_list<e>-objects

Initialization of `std::initializer_list<E>` objects

The underlying array of an `std::initializer_list<E>` is a **`const`** array of elements of type `E`, with length determined by the number of items in the braced list. Each element is copy initialized **??** — **??** on page **??** within Section 2.1."**??**" on page **??** by the corresponding expression in the braced list, and if user-defined conversions are required, they *must* be accessible at the point of construction. Following the rules of ***copy* list initialization**, narrowing conversions and explicit conversions are ill formed; see **??** — **??** on page **??** within Section 2.1."**??**" on page **??**:

```
struct X { operator int() const; };
void f(std::initializer_list<int>);

void testCallF()
{
    f({ 1, '2', X() });  // OK, 1 is int.
                         //     2 has a language-defined conversion to int.
                         //     X has a user-defined conversion to int.

    f({ 1, 2.5 });       // Error, 2.5 has a narrowing conversion to int.
}
```

Note that, since the initializer is a constant expression, narrowing conversions from integer literal constant expressions of a wider type are permitted:

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

constexpr long long lli = 13LL;
const     long long llj = 17LL;

void g(const long long arg)
{
    std::initializer_list<int> x = { 0LL };  // OK, integral constant
    std::initializer_list<int> y = { lli };  // OK, integral constant
    std::initializer_list<int> z = { llj };  // OK, integral constant
    std::initializer_list<int> w = { arg };  // Error, narrowing conversion
}
```

n-of-initializer_list

Type deduction of `initializer_list`

A `std::initializer_list` will not be deduced for a braced-initializer argument to a function template with an *unconstrained* template parameter, but a braced-initializer argument can be a match for a template parameter specifically declared as an `std::initializer_list<E>`. In such a case, the supplied list must not be empty, and the type deduced must be the same for each item of the list; otherwise the program is ill formed.

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

void f(std::initializer_list<int>);
template <typename E> void g(std::initializer_list<E>);
template <typename E> void h(E);

void test()
{
    f({ });                 // OK, empty list of int requires no deduction
    f({ 1, '2' });          // OK, all list initializers convert to int.

    g({ 1, 2, 3 });         // OK, std::initializer_list<int> deduced
    g({ });                 // Error, cannot deduce an E from an empty list
    g({ 1, '2' });          // Error, different deduced types

    h({ 1, 2, 3 });         // Error, cannot deduce an std::initializer_list
}
```

Note that the problem with an empty `std::initializer_list` when calling `g` is entirely due to the inability to deduce the type of the parameter; there is no problem passing an empty `std::initializer_list` to `f` since no type deduction is required. Similarly, heterogeneous lists can be passed to a known instantiation of `std::initializer_list` as long as all supplied list elements are implicitly convertible to the element type of the list.

Unlike an unconstrained template argument, an **`auto`** variable (see Section 2.1."**??**" on page **??**) will deduce an `std::initializer_list` in *copy* list initialization. However, to resolve potential ambiguity in favor of a copy constructor, ***direct* list initialization** of an **`auto`** variable requires a list of exactly one element and is deduced to be a copy of the same type:

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

auto a = {1, 2, 3};  // OK, a is std::initializer_list<int>.
auto b{1, 2, 3};     // Error, too many elements in list
auto c = {1};        // OK, c is std::initializer_list<int>.
auto d{1};           // OK, d is int.
auto e = {};         // Error, cannot deduce element type from empty list
auto f{};            // Error, cannot deduce variable type from empty list
```

Note that the declarations of `b` and `d` were interpreted differently in the original C++11 Standard to both be valid and deduce as `std::initializer_list<int>`. However, this behavior was changed in C++17 with N3922 and applied retroactively to previous standards as a defect report. Early C++11 compilers may still display the original behavior, although most compilers managed to adopted the DR before completing their C++14 implementation.

d-std::initializer_list

Overload resolution and `std::initializer_list`

When name lookup finds an overload set containing `std::initializer_list` arguments, the ranking rules now prefer to match braced lists to the `std::initializer_list` overloads rather than interpret them as other forms of list initialization, unless a default constructor would be selected; see Section 2.1."**??**" on page **??**.

use-cases-initlist

Use Cases

-of-standard-containers

Convenient population of standard containers

The original design intent of `std::initializer_list` was to make initializing a container with a sequence of values as simple as initializing a built-in array:

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

std::vector<int> v = {1, 2, 3, 4, 5};
```

Note that because `std::vector` deliberately intends to make available this form of initialization, the `<vector>` header is required to transitively `#include <initializer_list>`, so that the user does not need to. This is the case for the header of any standard library container, which means we do not generally see explicit inclusion of the `<initializer_list>` header in application code, since most uses are through standard containers.

As users start to think of braced lists as stand-ins for containers, it was naturally expected that any standard container with a `std::initializer_list` constructor would also be assignable from a braced list, so all such containers also provide an overload for the assignment operator, to avoid implicitly creating a temporary to pass to the move-assignment operator:

```
void test1()
{
    v = {2, 3, 5, 7, 11, 13, 17};
}
```

Finally, because `std::initializer_list` satisfies the requirements of a random access range, all container member function templates that take a pair of iterators to denote a range were overloaded to also accept an `std::initializer_list` of the corresponding type:

```
void test2()
{
    v.insert(v.end(), {23, 29, 31});
}
```

Note that since the rules for braced initialization are recursive, more complex containers can also use this syntax, which may be especially useful when creating **const** containers since there is no other way to provide the initial data without making another copy with the same lifetime as the container itself:

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

const std::map<std::string, int> m{{"a", 1}, {"b", 2}, {"c", 3}};
```

Providing support for braced lists in our own class

ists-in-our-own-class

Just as the C++ Standard Library provides extensive support for supplying braced lists to containers, so our client will come to expect the same of the classes we provide. For this example, we create a class that holds a dynamic array of **int**s that can be initialized and set to an arbitrary list of values supplied by the user. To simplify the example, we implement in terms of `std::vector` to avoid the distracting complexity of explicit memory management; in practice, the intent of such a class would be to provide a type simpler than `std::vector` itself:

```
    std::vectorassertstd::memcmp

class DynArray
{
    // DATA
    std::vector<int> d_data;

    // FRIENDS
    friend bool operator==(const DynArray&            lhs,
                           std::initializer_list<int> rhs);

  public:
    // CREATORS
    DynArray() : d_data() {}
    DynArray(std::initializer_list<int> il) : d_data(il) {}

    DynArray(const DynArray&) = delete;
    DynArray& operator=(const DynArray&) = delete;

    // MANIPULATORS
    DynArray& operator=(std::initializer_list<int> il)
    {
        d_data = il;

        return *this;
    }

    DynArray& operator+=(std::initializer_list<int> il)
    {
        d_data.insert(d_data.end(), il);
```

```
            return *this;
        }

        void shrink(std::size_t newSize)
        {
            assert(newSize <= d_data.size());

            d_data.resize(newSize);
        }

        // ACCESSORS
        bool isEqual(std::initializer_list<int> rhs) const
        {
            // optimized with memcmpto take advantage of memory layout guarantees.
            return d_data.size() == rhs.size()
                && 0 == std::memcmp(d_data.data(),
                                    rhs.begin(),
                                    rhs.size() * sizeof(int));
        }
    };

    inline
    bool operator==(const DynArray& lhs, std::initializer_list<int> rhs)
    {
        return lhs.isEqual(rhs);
    }

    inline
    bool operator!=(const DynArray& lhs, std::initializer_list<int> rhs)
    {
        return !lhs.isEqual(rhs);
    }
```

There are many opportunities to provide rich support for braced lists beyond just the constructor and assignment operator. Let's exercise this class in `main`, highlighting its support for braced lists:

```
int main()
{
    DynArray x = { 1, 2, 3, 4, 5 };

    std::initializer_list<int> il = { 1, 2, 3, 4, 5 };

    assert(x == il);

    assert((x == { 1, 2, 3, 4, 5 }));  // Error, not a std::initializer_list
    assert((x != { 1, 2, 3 }));        // Error, not a std::initializer_list

    assert( x.isEqual({ 1, 2, 3, 4, 5 }));
    assert(!x.isEqual({ 1, 2, 3, 4, 6 }));
    assert(!x.isEqual({ 1, 2, 3, 4 }));
    assert(!x.isEqual({ }));

    x += { 6 };

    assert(x != il);
    assert( x.isEqual({ 1, 2, 3, 4, 5, 6 }));

    x.shrink(2);

    std::initializer_list<int> ilB = { 1, 2 };

    assert(!x.isEqual({ 1, 2, 3, 4, 5, 6 }));
    assert( x.isEqual({ 1, 2 }));
    assert( x.isEqual(ilB));
    assert(x == ilB);

    assert((x == { 1, 2 }));  // Error, not a std::initializer_list

    x += { 8, 9, 10 };

    assert(!x.isEqual({ 1, 2 }));
    assert( x.isEqual({ 1, 2, 8, 9, 10 }));
}
```

Note that due to a quirk of the C++ grammar for certain operators, using *braced-init-list* as the right-hand argument of an **operator**== does not implicitly deduce an *initializer-list*, even if the **operator**== takes an *initializer-list* as the parameter; see **??** — **??** on page **??** within Section 2.1."**??**" on page **??**.

Functions consuming a variable number of arguments of the same type

ents-of-the-same-type

Suppose we want a function that takes an arbitrary number of arguments all of the same type. In our example, we write a function that concatenates a number of input strings together separated by commas:

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

std::string concatenate(std::initializer_list<std::string> ils)
{
    std::string separator;
    std::string result;
    for (const std::string* p = ils.begin(); p != ils.end(); ++p)
    {
        result.append(separator);
        result.append(*p);
        separator = ",";
    }
```

```
        return result;
    }

    std::string hex_digits = concatenate({"A", "B", "C", "D", "E"});
```

An example from the C++ Standard Library would be the additional overload for `std::min` provided by C++11 to take a list of arguments:

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

constexpr int n = std::min({3,2,7,5,-1,3,9});  // min is constexpr in C++14.
static_assert(n == -1, "Error: wrong value?!");
```

ixed-number-of-objects

Iterating over a fixed number of objects

`std::initializer_list` was designed from the start to model a standard library range, which is the same set of requirements for a type to support the C++11 range-based for loop (see Section 2.1."**??**" on page **??**. Range-based **for** loops acquire their meaning through a translation equivalence; it is instructive to examine that as an example of the synergy of several new language features designed to interact well. Suppose that the compiler encounters the following range-based **for** loop:

```
for (declaration : { list }) statement
```

Such a loop would be translated as follows:

```
{
    auto &&__r = { list };
    for (auto __b = __r.begin(), __e = __r.end(); __b != __e; ++__b)
    {
        declaration = *__b;
        statement
    }
}
```

Note the use of **auto**&& to perfectly deduce the type of the argument and forward it into the following code by reference, avoiding the creation of unnecessary temporary objects.

Thus, `__r` is deduced to be a `std::initializer_list<E>`. By the special rule for creating `std::initializer_list` objects directly from a braced list and the corresponding lifetime extension rules for the reference `__r` to the deduced `std::initializer_list`, the lifetime of the array is extended to the lifetime of `__r`, which also encompasses the lifetime of the loop. Ultimately, that extension makes things work "as expected"; the array exists while the loop is executing and is destroyed immediately after.

Now suppose we want to iterate over a fixed set of objects. Using initializer lists lets us avoid the tedious boilerplate of creating our own array of data for the iteration. In our example, we analyze a character string to see if it begins with one of the numeric base prefix indicators[1]:

[1]To avoid unnecessary string copies, the standard C++17 `std::string_view` facility could be used in place of `std::string` in the `hasBasePrefix` example.

```
    std::initializer_liststd::string

bool hasBasePrefix(const std::string& number)
{
    for (const std::string& prefix : { "0x", "0X", "0b", "0B" })
    {
        if (number.find(prefix) == 0)
        {
            return true;
        }
    }

    return false;
}
```

Note that this pattern may often be seen feeding values to a test driver.

potential-pitfalls

Potential Pitfalls

ary-underlying-arrays

Dangling references to temporary underlying arrays

Because `std::initializer_list` has pointer semantics, it has all the pitfalls associated with regular pointers with respect to dangling references, i.e, referring to an object after its lifetime has ended. These problems arise when the lifetime of the `std::initializer_list` object is greater than the lifetime of the underlying array. As a simple example, the following risks undefined behavior and does not work as the user (probably) expects:

```
    std::initializer_list

void test()
{
    std::initializer_list<int> il;
    il = { 1, 2, 3 };  // BAD IDEA, bound to a temporary that is about to expire
}
```

In this example, the temporary array created for the brace-enclosed list does not have its lifetime extended, because `il` is being assigned, not initialized. If we are lucky, the compiler will warn us:

```
warning: assignment from temporary 'initializer_list' does not extend the lifetime
of the underlying array
```

As is so often the case when dealing with lifetime issues in C++, caution is required.

ances-initializerlist

Annoyances

omogeneous,-sometimes

Initializer lists must be homogeneous, sometimes

While a `std::initializer_list<E>` is clearly always homogeneous, the initializer list used to create it in many cases can be a heterogeneous list of initializers convertible to the common type `E`, but, in deduction contexts, the braced list must strictly be homogeneous too:

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

```
void f(std::initializer_list<int>) {}

template <typename E>
void g(std::initializer_list<E>) {}

int main()
{
    f({1, '2', 3});  // OK, heterogeneous list converts
    g({1, '2', 3});  // Error, cannot deduce heterogeneous list
    g({1,  2 , 3});  // OK, homogenous list

    auto x = {1, '2', 3};  // Error, cannot deduce heterogeneous list
    auto y = {1,  2 , 3};  // OK, homogenous list
    std::initializer_list<int> z = {1, '2', 3};  // OK, converts
}
```

icitly-declared-default

`std::initializer_list` constructor suppresses the implicitly declared default

The declaration of a constructor with an *initializer-list* argument will suppress the implicit declaration of a default constructor, as one would expect. Without a default constructor, the `std::initializer_list` constructor will be called when the object is initialized from an empty list but not in other circumstances where a default constructor might be called. If such a type is then used as the type of a subobject, it would result in an implicitly declared default constructor of the outer object being deleted. These rules can make initialization from a pair of empty braces a bit counter-intuitive:

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

struct X
{
    std::vector<int> d_v;

    X(std::initializer_list<int> il) : d_v(il) {}
};

struct Y
{
    long long d_data;

    Y() : d_data(-1) {}
    Y(std::initializer_list<int> il) : d_data(il.size()) {}
};

struct Z1 : X { };

struct Z2 : X
```

```
{
    Z2() = default;         // BAD IDEA, implicitly deleted
};

struct Z3 : X
{
    using X::X;
};

struct Z4
{
    X data;

    Z4() = default;
};

void demo()
{
    X a;                    // Error, no default constructor

    X b{};                  // OK
    assert(b.d_v.empty());

    X c = {};               // OK
    assert(c.d_v.empty());

    X d = { 1, 2, 7 };      // OK
    assert(3 == d.d_v.size() && 7 == d.d_v.back());

    X ax[5];                // Error, no default constructor
    X bx[5]{};              // OK, initializes each element with {}
    X cx[5] = {};           // OK, initializes each element with {}

    Y e;                    // OK
    assert(-1 == e.d_data);

    Y f{};                  // OK, default constructor!
    assert(-1 == f.d_data);

    Y g = {};               // OK, default constructor!
    assert(-1 == g.d_data);

    Y h({});                // OK, using std::initializer_list
    assert(0  == h.d_data);

    Y i = { 7, 8, 9 };      // OK, using std::initializer_list
    assert(3  == i.d_data);

    Z1 j1{};       // Error, calls implicitly deleted default constructor
    Z2 j2{};       // Error, calls deleted default constructor
    Z3 j3{};       // Error, calls implicitly deleted default constructor
    Z4 j4{};       // OK, aggregate initialization, empty list for X

    Z1 k1 = {};    // Error, calls implicitly deleted default constructor
    Z2 k2 = {};    // Error, calls deleted default constructor
    Z3 k3 = {};    // Error, calls implicitly deleted default constructor
    Z4 k4 = {};    // OK, aggregate initialization, empty list for X

    Z1 m1 = {{}};  // Error, not an aggregate, no initializer_list constructor
    Z2 m2 = {{}};  // Error, not an aggregate, no initializer_list constructor
    Z3 m3 = {{}};  // OK, initializer_list constructor with one int
    Z4 m4 = {{}};  // OK, aggregate initialization, empty list for X
}
```

Note that most of the `Z*` examples will become "OK" in C++17 as the definition of aggregate evolves. Some of those examples will become errors again in C++20 as will all of the `Z4` examples above because `Z4` ceases to be an aggregate in C++20 due to the user-declared default constructor.

On the other hand, with regard to nondefault constructors, if a nonempty pair of braces is called, the *initializer-list* constructor is favored over all others; see **??** — **??** on page **??** within Section 2.1."**??**" on page **??**:

```
    assertstd::vector

void test()
{
    std::vector<int> v{ 12, 3 };
    assert(2 == v.size() && v.front() == 12);
    std::vector<int> w( 12, 3 );
    assert(12 == w.size() && w.front() == 3);
}
```

Initializer lists represent `const` objects

represent-const-objects

The underlying array of an initializer list is **`const`**, so the objects in that list cannot be modified. In particular, it is not possible to *move* elements out of the list that are typically consumed just once, despite their being temporaries.

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

std::vector<std::vector<int>> v = {{1, 2, 3},
                                   {4, 5, 6},
                                   {7, 8},
                                   {9}};
```

In this example, the vector, `v`, is initialized by an *rvalue* that is an `std::initializer_list` comprising four temporary vectors, each allocating memory for their internal array. However, the constructor for `v` must again copy each vector, allocating a fresh copy of each array, rather than take advantage of move-related optimizations, since each temporary is effectively **`const`**-qualified.

Another consequence of initializer lists' **const** nature is that *move-only* types such as std::unique_ptr<T> become completely unusable with them.

are-largely-vestigial

Overloaded free-function templates begin and end are largely vestigial

The range-based **for** feature initially required the free functions begin and end; see *Appendix* on page 22 for why they are no longer needed. The begin and end free functions were already provided by the <iterator> header, but because the <initializer_list> header is freestanding unlike the <iterator> header, it was necessary to also add specific overloads to the <initializer_list> header. Historically, due to the perceived range-based **for** dependency, the aforementioned free functions in the <iterator> header were also added to many other headers, including those for all of the standard containers.

As subsequent editions of the Standard have added to the set of related, free function overloads (for example, empty, added in C++17), those additional overloads have been added to multiple headers so that idiomatic generic code does not need to include additional headers. However, due to the freestanding requirement, none of those additional overloads are available through the <initializer_list> header.

see-also

See Also

- "??" (§2.1, p. ??) ♦ provides further details regarding object initialization and construction using braced lists and std::initializer_lists.

further-reading

Further Reading

- ?[https://www.stroustrup.com/N1890-initialization.pdf]
- ?[https://www.stroustrup.com/N1919-initializer_lists.pdf]
- ?

appendix-initlist

Appendix

pport-range-based-for

A brief history of user customization to support range-based for

The original incarnation of the range-based **for** loop was built around the proposed **concepts** language feature. The range argument to the **for** loop would satisfy the range concept, which the user could customize with a concept_map to adapt third-party libraries that did not have begin and end member functions.

When concepts were pulled after the first ISO ballot, the committee did not want to lose range-based **for**, so a new scheme was invented: The compiler would look for begin and end free functions. The Standard Library would provide the primary template for these functions, and **#include** <iterator> would be required for the range-based **for** loop to work, much like **#include** <typeinfo> is required to enable the **typeid** operator. Suggestions were made — and rejected — that the core feature should look for the member functions (just like the template) before looking for the free functions as the last resort. To make the standard library easy to use, the primary template for the begin and end functions was added to every standard container header, to <regex>, to <string>, and would be added on an ongoing basis to any new headers for types representing a range.

The Core Working Group then met the original request, so that range-based **`for`** never uses the `<iterator>` templates but directly looks for member functions and, failing that, performs an **ADL** lookup to find `begin` and `end`. The need to scatter these templates in every header is gone, but that core language change was not communicated to the Library Working Group in time. The only reason these functions remain in so many headers today is backward compatibility from when they were redundantly specified in C++11.

While the `begin` and `end` free functions are never required to support the range-based **`for`** loop, subsequent evolution of the Standard Library has embraced these functions as a means of expressing generic code. When called in a context that supports ADL, arbitrary third party types can be adapted to satisfy the requirements of a Standard Library range. Further overloads for constant and reverse iterators were added for C++14 and for a broader range of queries, such as `empty` and `size`, in C++17. Unfortunately, due to the way the Standard Library specification is drafted, all of these additional overloads are scattered across the same set of additional headers, despite being totally unrelated to the vanished motivation of supporting range-based **`for`**.

sec-conditional-cpp14

Chapter 3

Unsafe Features

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

sec-unsafe-cpp14

Deducing Types Using decltype Semantics

sing-decltypedecltypeauto

In a C++14 variable declaration, **decltype(auto)** can act as a **placeholder type** that is deduced to exactly match the type of the variable's initializer, preserving the initializer's **value category**, unlike the **auto** placeholder.

Description

description

The type specifier, **auto** (see Section 2.1."**??**" on page **??**), can be used in C++11 as a placeholder to declare a variable whose type is deduced from the variable's initializer:

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

C f1();

auto a1 = 0;    // deduced type int
auto a2{f1()};  // deduced type C
```

C++14 introduced a new placeholder, **decltype(auto)**, which can be used in most of the same contexts as **auto**. For the example above, **decltype(auto)** behaves identically to **auto**:

```
decltype(auto) a3 = 0;    // deduced type int
decltype(auto) a4{f1()};  // deduced type C
```

The literal, `0`, has type **int**; initializing `a3` with `0` thus yields a variable of type **int**. Similarly, the expression `f1()` has type `C`, yielding a variable of type `C` when used to initialize `a4`.

Unlike plain **auto**, the deduced type of a variable declared with type **decltype(auto)** is determined not by using template-argument deduction rules, but by applying the **decltype** operator to the initialization expression. In practice, this semantic means that the cv-qualifiers and value category (see Appendix **??** on page **??**) of the initializer are preserved for **decltype(auto)** when they would be discarded for plain **auto**:

```
int&    f2();
C&&     f3();
C       c1;
const C cc1;
C*      p1 = &c1;

auto           i1  = 4;      // deduced as int
decltype(auto) i2  = 4;      //    "     " int

auto           c7  = f2();   //    "     " int
decltype(auto) c8  = f2();   //    "     " int&

auto           i3  = f3();   //    "     " C
decltype(auto) i4  = f3();   //    "     " C&&
auto           c9  = cc1;    //    "      " C
decltype(auto) c10 = cc1;    //    "      " const C

auto           c11 = (cc1);  //    "      " C
decltype(auto) c12 = (cc1);  //    "      " const C&
```

As with the **decltype** operator, **decltype(auto)** is one of the few constructs that preserves the distinction between the two categories of *rvalue* — *prvalues* (e.g., literal `4`) and *xvalues* (e.g., `f3()`).

Both **auto** and **decltype(auto)** can be used as placeholders for a function return type, indicating that the return type should be deduced from the function's **return** statement(s). The deduced return-type feature is covered in its own section (see Section 3.2."**??**" on page **??**). Note that the deduction rules described for function return types in that section refer to the ones described for variables here. Readers are, therefore, advised to read this section first.

Specification

specification

The **decltype(auto)** placeholder can appear in most places where **auto** can appear:

1. As the type in the declaration of an initialized variable, including variables defined in the header of a loop or **switch** statement
2. As the type of object allocated and initialized by a **new** expression
3. As the type returned by a function or conversion operator

The last is the most common use of **decltype(auto)** and is described in detail in Section 3.2."**??**" on page **??**. Note that **decltype(auto)** cannot be used to declare a parameter of a generic lambda expression; see Section 2.2."**??**" on page **??**.

For a variable, `v`, declared with **decltype(auto)** and initialized with an expression, `expr`, the type of `v` is deduced to be the type denoted by **decltype**`(expr)`; see Section 1.1."**??**" on page **??**. This semantic means that the deduced type might be cv-qualified and/or a reference:

```
struct C1 { /*...*/ };

int&     f2();
C1&&     f3();
C1       c1;
const C1 cc1;

decltype(f2())  i1  = f2();   // deduced as int&
decltype(auto)  i2  = f2();   // equivalent to i1

decltype(f3())  c2  = f3();   // deduced as C1&&
decltype(auto)  c3  = f3();   // equivalent to c2

decltype(c1)    c4  = c1;     // deduced as C1
decltype(auto)  c5  = c1;     // equivalent to c4
decltype((c1))  c6  = c1;     // deduced as C1&
decltype(auto)  c7  = (c1);   // equivalent to c6

decltype(cc1)   cc2 = cc1;    // deduced as const C1
decltype(auto)  cc3 = cc1;    // equivalent to cc2
decltype((cc1)) cc4 = cc1;    // deduced as const C1&
decltype(auto)  cc5 = (cc1);  // equivalent to cc4

decltype({ 3 }) x1 = { 3 };   // Error, not an expression
decltype(auto)  x2 = { 3 };   // Error, not an expression
```

The semantics of the **decltype** operator, when applied to an expression consisting of a single variable, cause **decltype**(c1) to yield type C1 and **decltype**((c1)) to yield reference type C1&, as in the definitions of c4 and c6, respectively; variables c5 and c7, therefore, also have the types C1 and C1&. A braced-initializer list such as { 3 } is not an expression; thus, x1 and x2 are both invalid.

Note that functions returning scalars discard top-level cv-qualifiers on their return types, so a type deduced from a call to such a function will not reflect top-level cv-qualifiers even when defined with **decltype**(**auto**):

```
template <typename T> T f4();

decltype(auto) v1 = f4<const C1>();           // deduced as const C1
decltype(auto) v2 = f4<const int>();          //    "     " int
decltype(auto) v3 = f4<const int&>();         //    "     " const int&
decltype(auto) v4 = f4<const char* const>();  //    "     " const char*
```

The top-level **const** qualifier on the class type, **const** C1, and on the reference type, **const int**&, are preserved but not on the scalar type, **const int**. The **const**ness of the pointer itself, in **const char*** **const**, is similarly discarded, as it is the top-level cv-qualifier on a scalar type.

When a function name is used as the initializer expression, it automatically decays to a pointer type when initializing a variable declared with type **auto** but does not decay when the variable is declared with type **decltype**(**auto**); thus, the deduced type of fx2 is a function type, which is not an allowed type for a variable. Initializer expressions having pointer-to-function type or reference-to-function type do not pose a problem, as seen with fx3 and fx4.

```
auto           fx1 = f2;     // OK, deduced as (decayed type) int& (*)()
decltype(auto) fx2 = f2;     // Error, cannot define variable of type int&()

auto           fx3 = &f3;    // OK, deduced as C1&& (*)()
decltype(auto) fx4 = &f3;    // OK,    "     " C1&& (*)()

auto&          fx5 = *fx3;   // OK,    "     " C1&& (&)()
decltype(auto) fx6 = *fx3;   // OK,    "     " C1&& (&)()
```

Note that fx5 uses & to force the variable type to be a reference. The ability to add reference specifiers and cv-qualifiers to a placeholder is not available for **decltype**(**auto**), as described in the next section.

Syntactic restrictions

yntactic-restrictions

When used as a type placeholder, **decltype(auto)** must appear alone, unadorned by cv-qualifiers, reference type specifiers, pointer type specifiers, or function parameter lists:

```
int&& f1();
int i1 = 5;

decltype(auto)       i2     = f1();  // OK, deduced as int&&
const decltype(auto) i3     = f1();  // Error, const qualifier not allowed
decltype(auto)&&     i4     = f1();  // Error, reference operator "     "
decltype(auto)*      i5     = &i1;   // Error, pointer operator   "     "
decltype(auto)     (*fx1)() = &f1;   // Error, function parameters "    "
```

All of the above definitions would be valid if **decltype(auto)** were replaced with **auto**:

```
auto                 i6     = f1();  // OK, i6 deduced as int
const auto           i7     = f1();  // OK, i7    "     " const int
auto&&               i8     = f1();  // OK, i8    "     " int&&
auto*                i9     = &i1;   // OK, i9    "     " int*
auto               (*fx2)() = &f1;   // OK, fx2   "     " int&& (*)()
```

The **decltype(auto)** placeholder cannot be used to define a variable without an initializer because there would be no way to deduce its type. If multiple variables are defined in a single **decltype(auto)** definition, they must all have initializers of exactly the same type:

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

decltype(auto) v1;                             // Error, no initializer
decltype(auto) v2 = f1(), v3 = std::move(i1);  // OK, deduced as int&&
decltype(auto) v4 = 5, v5 = f1();              // Error, ambiguous deduction
```

A non**static** member variable cannot be declared using **decltype(auto)**, even if provided with a default member initializer (see Section 2.1."**??**" on page **??**):

```
struct C1
{
    decltype(auto) d_data = f();  // Error, decltype(auto) for member variable
};
```

A **constexpr** static member variable (see Section 2.1."**??**" on page **??**) that is initialized at the point of declaration can be declared using **decltype(auto)**, but a non**constexpr** static member variable cannot, simply because non**constexpr** static members cannot be inline-initialized at the point of declaration, independently of the **decltype(auto)** feature:

```
constexpr int f2() { return 5; }

struct C2
{
    static constexpr decltype(auto) s_mem1 = f2();  // OK
    static           decltype(auto) s_mem2 = f2();  // Error, inline init
};
```

A variable with static storage duration (either in global or class scope) can be declared using an explicit type then redeclared and initialized using **decltype(auto)**. Note, however, that some popular compilers reject these redeclarations[1]:

```
extern int gi;  // forward declaration

struct C3
{
    static decltype(f2()) s_mem1;  // type int
};

decltype(auto) gi =         f2();  // OK, compatible redeclaration
decltype(auto) C3::s_mem1 = f2();  // OK, compatible redeclaration
```

new-expressions

new expressions

When used in a **new** expression, **decltype(auto)** offers little benefit over plain **auto** and will sometimes cause otherwise-valid code to not compile:

```
int   i;
int&& f1();

auto* p1 = new auto(5);                // OK, equivalent to new int(5)
auto* p2 = new decltype(auto)(5);      // OK, equivalent to new int(5)

auto* p3 = new auto(i);                // OK, equivalent to new int(i)
auto* p4 = new decltype(auto)(i);      // OK, equivalent to new int(i)

auto* p5 = new auto(f1());             // OK, equivalent to new int(f1())
auto* p6 = new decltype(auto)(f1());   // Error, equivalent to new int&&(f1())

auto* p7 = new auto((i));              // OK, equivalent to new int(i)
auto* p8 = new decltype(auto)((i));    // Error, equivalent to new int&(i)
```

In all of the examples above, the variable type is declared as **auto*** so that it could be deduced from the return type of the **new** expression; plain **auto**, **decltype(auto)**, or **int*** would all have been equivalent. The initializers for p6 and p8 fail to compile because **decltype(auto)** deduces to a reference type in each of those cases, causing the **new** expression to generate a pointer-to-reference type. The **auto** specifiers used to initialize p5 and p7, conversely, discard the reference qualifiers, yielding valid types.

use-cases

Use Cases

ype-and-value-category

Exact capture of an expression's type and value category

Both **auto&&** and **decltype(auto)** can be used to declare a variable initialized to the result of any expression, but only **decltype(auto)** will capture the exact value category of

[1]GCC 10.2 and MSVC 19.28, among many other compilers, reject **auto** redeclaration of previously declared variables; see GCC bug report 60352. However, nothing in the C++14 Standard appears to disallow such redeclarations, and an example in the C++20 Standard indicates that they are valid.

initializing expression:

```
int   f1();
int&  f2();
int&& f3();
int   i;

auto&&         v1 = f1();  // type int&&
decltype(auto) v2 = f1();  // type int

auto&&         v3 = f2();  // type int&
decltype(auto) v4 = f2();  // type int&

auto&&         v5 = f3();  // type int&&
decltype(auto) v6 = f3();  // type int&&

auto&&         v7 = i;     // type int&
decltype(auto) v8 = i;     // type int
```

Variables `v1` and `v5` have the same value category even though `f1()` is a *prvalue* and `f3()` is an *xvalue*, illustrating the limitation of **`auto`**`&&`, whereas `v2` and `v6` correctly capture the distinction. In addition, **`auto`**`&&` deduces `v7` as a reference whereas **`decltype`**`(`**`auto`**`)` correctly deduces it as an object.

or-or-moving-iterator

Return type of a proxy iterator or moving iterator

When an iterator is dereferenced, it usually returns an *lvalue* reference to an element within a sequence. It might, however, return an *rvalue* object of proxy type, as in the case of `std::vector<`**`bool`**`>`. Alternatively, the iterator dereference operator might return an *rvalue* reference to a sequence element, as in the case of a *moving iterator* — i.e., for a sequence that will not be used again after it has been traversed. **`decltype`**`(`**`auto`**`)` can be used in generic code to faithfully capture a dereferenced iterator when the value category of the iterator's `reference` type is unknown:

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

template <typename C, typename V>
void fill(C& container, const V& val)
    // Replace the value of every element in container with a copy of val.
{
    for (typename C::iterator iter = container.begin();
         iter != container.end();
         ++iter)
    {
        // auto& element = *iter;  // won't work for proxy or moving iterators
        decltype(auto) element = *iter;
        element = val;
    }
}
```

```
void f1(std::vector<bool>& v)
{
    fill(v, false);
    // ...
}
```

Instead of **decltype**(**auto**), we could have used **auto**&& and gotten the same effect, although the semantics of **decltype**(**auto**) are slightly simpler to understand in this case. Although more verbose, it would be more descriptive to declare `element` as having type **typename** `std::iterator_traits<`**decltype**`(iter)>::reference`.

potential-pitfalls

Potential Pitfalls

den-dangling-references

Hidden dangling references

An operation on an *rvalue* will sometimes return an *rvalue* reference that is valid only for the lifetime of the original *rvalue*. When the returned reference is saved in a named variable, there is a danger of the variable binding to a temporary object that will go out of scope before it can be used:

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

template <typename T>
T&& first(std::list<T>&& s) { return std::move(*s.begin()); }
    // Return an rvalue reference to the first element in s.

std::list<int> collection();
    // Return (by value) a list of int values.

void f()
{
    int&&          r1 = first(collection());
    auto&&         r2 = first(collection());
    decltype(auto) r3 = first(collection());

    // Bug, r1, r2, and r3 are all dangling references to destroyed
    // objects.

    // ...
}
```

The variables `r1`, `r2`, and `r3` all have type **int**&& and all are dangling references because they refer to an element of a `list` that goes out of scope immediately after the reference is initialized. When a reference variable is initialized from a reference expression, the programmer should be wary of the lifetime of the object being referenced. In this regard, **decltype**(**auto**) adds no new hazard. Note, however, that `r1` and `r2` are both *declared* as reference types, whereas `r3` is only *deduced* to be a reference type. The fact that the reference is *hidden* makes this pitfall harder to avoid for **decltype**(**auto**) than for the other two cases.

r-signaling-of-intent

Poor signaling of intent

Since C++11, a common set of idioms has emerged for the use of **auto** in generic code:

```
auto        copyVar     = expr1;  // copy of expr1
const auto& readonlyVar = expr2;  // read-only reference to expr1
auto&&      mutableVar  = expr3;  // possibly mutable reference to expr1
```

The `copyVar` object will be initialized from `expr1` using direct-initialization; if `expr1` yields an *rvalue* or *lvalue* reference, then the copy or move constructor, respectively, is invoked. The `readonlyVar` reference provides read-only access to the object produced by `expr2`; if `expr2` returns an *rvalue*, then **lifetime extension** ensures that it remains valid until `readonlyVar` goes out of scope. Finally, `mutableVar` allows modifying or moving from `expr3` (unless `expr3` is **const**); as in the case of **const auto**&, lifetime extension might come into play. Faithful use of these idioms provides safety and signals the programmer's intent regarding the expected use of the variable.

There is no way to create a similar set of idioms for **decltype(auto)** because **decltype(auto)** cannot be combined with **const** or reference type specifiers; see *Annoyances* — **decltype(auto)** *stands alone* on page 34. For variable declarations, therefore, using **auto** in this idiomatic way might be preferable.

The situation is somewhat different for *function return* types, where the expected use of the return value is not always known at the point of declaration; see Section 3.2."**??**" on page **??**.

noyances-decltypeauto

Annoyances

pe(auto)-stands-alone

decltype(auto) stands alone

When defining a variable with **auto**, we can add cv-qualifiers and reference type specifiers to the deduced type, even if the initializer expression has simpler qualifiers:

```
int f();

const auto& v1 = f();  // v1 is const int&.
```

Unfortunately, **decltype(auto)** must stand alone; the type of the variable is always exactly that of the initializer expression, with no extra decorations:

```
const decltype(auto) v2 = f();  // Error, const with decltype(auto)
```

Thus, deducing a variable type that is always read-only, for example, is not possible with **decltype(auto)**.

see-also

See Also

- "**??**" (§1.1, p. **??**) ♦ describes the C++11 feature on which **decltype(auto)** is based.
- Appendix **??** describes complex world of value categories that distinguish **decltype(auto)** from **auto**.
- "**??**" (§3.2, p. **??**) ♦ uses the deduction rules described in this section and applies them to function return types instead of variables declarations.

- "**??**" (§1.1, p. **??**) ♦ describes the **decltype** operator, which determines the semantics of **decltype(auto)**.

Further Reading

further-reading

None.