```
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-based **for** loop only because **lifetime 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 **for** loop too would have been considered to have undefined behavior; again, see *Pointer semantics and lifetimes of temporaries* below.

Finally, corresponding global `std::begin` and `std::end` **free function** templates are **overloaded** for `std::initializer_list` objects directly in the `<initializer_list>` header, but see *Annoyances — Overloaded free-function templates `begin` and `end` are largely vestigial* on page 570.

Pointer semantics and lifetimes of temporaries

An instance of the `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. For example, `std::initializer_list` might be implemented as a pair of pointers or a pointer and a length.

When a nonempty braced list is used to initialize an `std::initializer_list`, the compiler generates a temporary array having the same lifetime as other temporary objects created in the same **expression**. 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 an `std::initializer<int>` initialized with three values, `1`, `2`, and `3`:

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

The compiler first creates a temporary array holding the three values. That array would normally be destroyed at the end of the **outermost expression** in which it appears, but initializing `il` to refer to this array extends its lifetime to be coterminous with `il`.

No such lifetime extension occurs under any other circumstances:

```
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` having a dangling reference to an array that no longer exists; see *Potential Pitfalls — Dangling references to temporary underlying arrays* on page 566.

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** 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 Section 2.1."Braced Init" on page 215:

```
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, 2.0 });       // Error, 2.0 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, provided that they are lossless:

```
#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> v = { 1ll<<32 };  // Error, narrowing conversion
    std::initializer_list<int> w = { arg };      // Error, narrowing conversion
}
```

Type deduction of `initializer_list`

An `std::initializer_list` will not be deduced for a braced-initializer argument to a **function template** having an *unconstrained* **template parameter**, but a braced-initializer