```
// 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 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 compiler-supplied arrays. When these proxy objects are copied or assigned, they do not copy the elements of their underlying array. Note that `std::initializer_list` satisfies the Standard Library requirements of a range with random access iterators.

The public 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."**`constexpr`** Functions" on page 257. The **`noexcept`** specifier indicates that the function is not allowed to throw an exception; see Section 3.1. "**`noexcept`** Specifier" on page 1085.

As an introductory example, consider a function, `printNumbers`, that prints 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 passing arguments 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 invoke this `printNumbers` function on a **braced-initializer list**; see Section 2.1."Braced Init" on page 215: