

Subroutines and Control Abstraction

8.6.3 Implementation of Iterators

EXAMPLE 8.79

Coroutine-based iterator
invocation

Consider the following `for` loop from Example 6.63 (page 263):

```
for i in range(first, last, step):  
    ...
```

A compiler might translate this as

```
iter := new from_to_by(first, last, step, i, done, current_coroutine)  
while not done do  
    ...  
    transfer(iter)  
destroy(iter)
```

After the loop completes, the implementation can reclaim the space consumed by `iter`. ■

EXAMPLE 8.80

Coroutine-based iterator
implementation

The definition of `from_to_by` itself is quite straightforward:

```
coroutine from_to_by(from_val, to_val, by_amt : int;  
    ref i : int; ref done : bool; caller : coroutine)  
i := from_val  
if by_amt > 0 then  
    done := from_val ≤ to_val  
    detach  
    loop  
        i += by_amt  
        done := i ≤ to_val  
        transfer(caller)    -- yield i
```

```

else
  done := from_val ≥ to_val
  detach
  loop
    i += by_amt
    done := i ≥ to_val
    transfer(caller)    -- yield i

```

Parameters *i* and *done* are passed by reference so that the iterator can modify them in the caller's context. The caller's identity is passed as a final argument so that the iterator can tell which coroutine to resume when it has computed the next loop index. Because the caller is named explicitly, it is easy for iterators to nest, as in Figure 6.5 (page 264). ■

Single-Stack Implementation

While coroutines suffice for the implementation of iterators, they are not *necessary*. A simpler, single-stack implementation is also possible. Because a given iterator (e.g., an instance of `from_to_by`) is always resumed at the same place in the code (at the top of a given `for` loop), we can be sure that the subroutine call stack will always contain the same frames whenever the iterator runs. Moreover, since `yield` statements can appear only in the main body of the iterator (never in nested routines), we can be sure that the stack will always contain the same frames whenever the iterator transfers back to its caller. These two facts imply that we can place the frame of the iterator directly on top of the frame of its caller in a single central stack.

When an iterator is created, its frame is pushed on the stack. When it yields a value, control returns to the `for` loop, but the iterator's frame is left on the stack. If the body of the loop makes any subroutine calls, the frames for those calls will be allocated beyond the frame of the iterator. Since control must return to the loop before the iterator resumes, we know that such frames will be gone again before the iterator has a chance to see them: if it needs to call subroutines itself, the stack above it will be clear. Likewise, if the iterator calls any subroutines, they will return (popping their frames from the stack) before the `for` loop runs again. Nested iterators present no special problems (see Exercise 8.45).

Data Structure Implementation

Compilers for C# 2.0 employ yet another implementation of iterators. Like Java, C# 1.1 provided iterator objects. Each such object implements the `IEnumerator` interface, which provides `MoveNext` and `Current` methods. Typically an iterator is obtained by calling the `GetEnumerator` method of an object (a container) that implements the `IEnumerable` interface:

EXAMPLE 8.81

Iterator usage in C#

```

for (IEnumerator i = myTree.GetEnumerator(); i.MoveNext();) {
  object o = i.Current;
  Console.WriteLine(o.ToString());
}

```

EXAMPLE 8.82Implementation of C#
iterators

C# 2.0 provides true iterators as an extension of iterator objects. The programmer simply declares a method that contains one or more `yield return` statements, and whose return type is `IEnumerator` or `IEnumerable`. Here is an example of the latter:

```
static IEnumerable FromToBy (int fromVal, int toVal, int byAmt)
{
    if (byAmt >= 0) {
        for (int i = fromVal; i <= toVal; i += byAmt) {
            yield return i;
        }
    } else {
        for (int i = fromVal; i >= toVal; i -= byAmt) {
            yield return i;
        }
    }
}
```

The compiler automatically transforms this code into a hidden class with a `GetEnumerator` method, along the lines of Figure ©8.17. Within this code, an explicit state variable keeps track of the “program counter” of the last `yield` statement. In addition, local variable `i` of the true iterator becomes a data member of the `FromToByImpl` class, leaving the iterator with no need for a stack frame across iterations of the loop. In a quite literal sense, the compiler transforms each true iterator into an iterator object. ■

Recursive iterators present no particular difficulties: a nested iterator is allocated on demand when the outer iterator enters a `foreach` loop, and is referred to by a reference in that outer iterator. The details are deferred to Exercise ©8.46. Because iterator objects are allocated from the heap, the C# implementation of true iterators may be somewhat slower than the stack-based implementation of the previous subsection.

✓ CHECK YOUR UNDERSTANDING

77. Describe the “obvious” implementation of iterators using coroutines.
78. Explain how the state of multiple active iterators can be maintained in a single stack.
79. Describe the transformation used by C# compilers to turn a true iterator into an iterator object.

```

static IEnumerable FromToBy(int fromVal, int toVal, int byAmt) {
    return new FromToByImpl(fromVal, toVal, byAmt);
}
class FromToByImpl : IEnumerator, IEnumerable {
    enum State {starting, goingUp, goingDown, done}
    int i, tv, ba;
    State s;

    public FromToByImpl(int fromVal, int toVal, int byAmt) {
        i = fromVal; tv = toVal; ba = byAmt; s = State.starting;
    }
    public IEnumerator GetEnumerator() {
        return this;
    }
    public object Current {
        get { return i; }
    }
    public bool MoveNext() {
        switch (s) {
            case State.starting :
                if (ba >= 0) {
                    if (i <= tv) { s = State.goingUp; return true; }
                    else { s = State.done; return false; }
                } else {
                    if (i >= tv) { s = State.goingDown; return true; }
                    else { s = State.done; return false; }
                }
            case State.goingUp :
                i += ba;
                if (i <= tv) return true;
                else { s = State.done; return false; }
            case State.goingDown :
                i -= ba;
                if (i >= tv) return true;
                else { s = State.done; return false; }
            default: // for completeness
                case State.done : return false;
        }
    }
    public void Reset() {
        s = State.starting;
    }
}

```

Figure 8.17 Iterator object equivalent of a true iterator in C#. This handwritten code corresponds to Example 8.82. It represents, at the source level, what the compiler creates at the level of intermediate code: a state machine that tracks the program counter of the original iterator, with a starting state, an ending state, and one state for each **yield return** statement. The arms of the **switch** statement capture the code paths in the original iterator that move from one state to the next.