CHAPTER 23

The Fibonacci Sequence

If liberty means anything at all, it means the right to tell people what they do not want to hear.

—George Orwell

Subtlety chases the obvious in a never-ending spiral and never quite catches it.

—Nero Wolfe

23.1 Introduction

Those who enjoy mathematical elegance may share my appreciation of the Fibonacci sequence and its associated relationship, the Golden Ratio. In this chapter we're going to look at how we might represent this mathematical sequence as a collection, in the form of an STL sequence class, and then consider whether it might be better represented as an iterator, before finally coming back to seeing how a range-limited sequence is the most discoverable representation.

Unlike the other STL extensions described in this book, this one does not derive from any libraries. It is entirely pedagogical. As such, I trust you'll bear with me in some of the less practicable fancies used to illuminate the STL extension issues covered. For those who prefer real examples, worry not, this is the only such fanciful example in the whole book.

23.2 The Fibonacci Sequence

The Fibonacci sequence is a series of numbers, starting with the pair 0 and 1, where the value of each element is calculated as the sum of the two preceding it. Hence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1,597, 2,584, 4,181, 6,765, and so on, ad infinitum.

The ratio of each entry in the series to its next tends toward an irrational constant, known as the Golden Ratio, whose value is approximately 1.61803398875. The Golden Ratio appears to crop up in all kinds of places in the universe, from the ratio of aesthetically pleasing picture frames to the twirls of conch shells to the dimensions of the Parthenon. If you've not come across it before, I recommend you check it out.

23.3 Fibonacci as an STL Sequence

My first instinct when thinking about how to represent a mathematical sequence was to use an STL-compliant sequence, as shown in Listing 23.1. As we'll see, however, this is not as nice a fit as we might think. Since this is a notional collection—there are no elements in existence anywhere—the enumeration of the values in the sequence is carried out in the iterator, an instance of

the member class `const_iterator`, whose element reference category is *by-value temporary* (Section 3.3.5).

Listing 23.1 `Fibonacci_sequence` **Version 1 and Its Iterator Class**

```
class Fibonacci_sequence
{
public: // Member Types
  typedef uint32_t  value_type;
  class             const_iterator;
  . . .
public: // Iteration
  const_iterator  begin() const
  {
    return const_iterator(0, 1);
  }
  const_iterator  end() const;
  . . .
};

class Fibonacci_sequence::const_iterator
  : public std::iterator< std::forward_iterator_tag
                        , Fibonacci_sequence::value_type, ptrdiff_t
                        , void, Fibonacci_sequence::value_type // BVT
                        >
{
public: // Member Types
  typedef const_iterator                  class_type;
  typedef Fibonacci_sequence::value_type  value_type;
public: // Construction
  const_iterator(value_type i0, value_type i1);
public: // Iteration
  class_type& operator ++();
  class_type operator ++(int);
  value_type operator *() const;
public: // Comparison
  bool equal(class_type const& rhs) const
  {
    return m_i0 == rhs.m_i0 && m_i1 == rhs.m_i1;
  }
  . . .
private: // Member Variables
  value_type  m_i0;
  value_type  m_i1;
};

inline bool operator ==(Fibonacci_sequence::const_iterator const& lhs
                      , Fibonacci_sequence::const_iterator const& rhs)
```

```
{
  return lhs.equal(rhs);
}
inline bool operator !=(Fibonacci_sequence::const_iterator const& lhs
                      , Fibonacci_sequence::const_iterator const& rhs)
{
  return !lhs.equal(rhs);
}
```

Listing 23.2 shows the implementations of the only two nonboilerplate methods of `const_iterator`.

Listing 23.2 Version 1: Preincrement and Dereference Operators

```
class_type& Fibonacci_sequence::const_iterator::operator ++()
{
  value_type  res   = m_i0 + m_i1;
              m_i0  = m_i1;
              m_i1  = res;
  return *this;
}
value_type Fibonacci_sequence::const_iterator::operator *() const
{
  return m_i0;
}
```

Each time the preincrement operator is called, the next result is calculated and moved into `m_i1`, after `m_i1` is first moved into `m_i0`. The current result is held in `m_i0`. Note that the `const_iterator` could just as easily support the *bidirectional* iterator category, wherein the predecrement operator would subtract `m_i0` from `m_i1` to get the previous value in the sequence. I've not done so simply because the Fibonacci is a forward sequence.

Because the sequence is infinite, `end()` is defined to return an instance of `const_iterator` whose value is such that it will never compare `equal()` to a valid iterator. (The implementation shown in Listing 23.3 corresponds to *`Fibonacci_sequence_1.hpp`* on the CD.)

Listing 23.3 Version 1: `end()` Method

```
class Fibonacci_sequence
{
  . . .
  const_iterator  end() const
  {
    return const_iterator(0, 0);
  }
  . . .
```

Let's now use this definition of the sequence:

```
Fibonacci_sequence                   fs;
Fibonacci_sequence::const_iterator   b = fs.begin();

for(size_t i = 0; i < 10; ++i, ++b)
{
  std::cout << i << ": " << *b << std::endl;
}
```

This works a treat, giving the first ten elements in the Fibonacci sequence: 0–34. However, as we well know, iterators like to work with algorithms and usually take them in pairs, for example:

```
std::copy(fs.begin(), fs.end()
    , std::ostream_iterator<Fibonacci_sequence::value_type>(std::cout
                                                            , " "));
```

Unfortunately, there are two problems with this statement. First, it runs forever, which represents somewhat of an inconvenience when you want to use your computer for something worthwhile, such as updating it with the latest virus definitions and operating system patches to fill up that last 12GB of disk you were saving for your database of fine European chocolatiers. You might wonder whether we will be saved when the overflowed arithmetic happens on a result whose value modulo 0x10000000 is 0. Although this does eventually occur—after 3,221,225,426 iterations, as it happens—the iterator still does not compare equal to the `end()` iterator because its `m_i1` member is nonzero. Since it is not possible for both members to be 0 at one time, the code will loop forever.

Second, after the forty-seventh iteration, the results returned are no longer members of the Fibonacci sequence but pseudo junk values as a consequence of overflow of our 32-bit value type. As we know, computers don't generally like to live in the infinite, and integral types are particularly antipathetic to unconstrained ranges.

23.3.1 Interface of an Infinite Sequence

We'll deal with the first problem first. Since the Fibonacci sequence is infinite, one option would be to make the `Fibonacci_sequence` infinite also. This is easily effected by removing the `end()` method. The sequence is now quite literally one without end. Now users of the class cannot make the mistake, shown earlier, of passing an ostensibly bounded [`begin()`, `end()`) range to an algorithm since there is no end.

In my opinion, this is the most appealing form from a conceptual point of view because the public interface of the sequence is representing its semantics most clearly. However, it's not terribly practical because, as we've already seen, overflow occurs after a soberingly finite number of steps. For infinite sequences whose values are bound within a representable range, this would be a good candidate approach, but it's not suitable for the Fibonacci sequence.

Note that this reasoning also rules out the possible alternative implementations of Fibonacci sequences as independent iterator classes or as generator functions.

23.3.2 Put a Contract on It

Let's now take the sensible step of putting some contract programming protection into the preincrement operator before we attempt to use the sequence. (The implementation shown in Listing 23.4 corresponds to *Fibonacci_sequence_2.hpp* on the CD.)

Listing 23.4 Version 2: Preincrement Operator

```
class_type& Fibonacci_sequence::const_iterator::operator ++()
{
  STLSOFT_MESSAGE_ASSERT("Exhausted integral type", m_i0 <= m_i1);
  value_type  res   = m_i0 + m_i1;
              m_i0  = m_i1;
              m_i1  = res;
  return *this;
}
```

In executing the `std::copy` statement shown previously, we find that the assertion is fired on the increment after output of the value 2,971,215,073. At this point, the previous value was 1,836,311,903, so we would expect `m_i1` to be 4,807,526,976. However, that exceeds the maximum value representable in a 32-bit unsigned integer (4,294,967,295), so the result is truncated (to 512,559,680), and the assertion fires. Hence, although we've managed to iterate 48 items, the last increment left the iterator in an invalid state, an unincrementable state, so there are only actually 47 viable enumerable values from a 32-bit representation.

I want to stress the distinction between providing a usable interface and guarding against misuse, well exemplified in this case. Thus far, our Fibonacci sequence does not have a usable interface—since its failure is a matter of surprise—but now, with the introduction of the assertion, it does have protection against its misuse.

23.3.3 Changing Value Type?

Perhaps a solution lies in using a different value type. Obviously, using `uint64_t` is only going to be a small bandage over the problem, allowing us to enumerate 93 steps and get to 7,540,113,804,746,346,429. And once we're there, we still precipitate a contract violation, indicating abuse of the sequence.

Maybe floating point is the way to go? (This implementation corresponds to *Fibonacci_sequence_3.hpp* on the CD.) Alas, no—32-bit `float` enters INF territory at 187 entries, 64-bit `double` at 1,478. Furthermore, since the entries in the sequence are not nicely rounded 10^N values, rounding errors creep in as soon as the exponent value reaches the extent of the mantissa.

Conceivably, a `BigInt` type using coded decimal evaluation would be able to go infinite, but it would have correspondingly poor performance. (Readers are invited to submit such a solution. In reward I can promise the unquantifiable fame that will come from having your name on the book's Web site.)

23.3.4 Constraining Type

To avoid floating-point inaccuracies, we would like to constrain the value type to be integral. To avail ourselves of the maximum range of the type and to catch overflow, we would like to constrain the value type to be unsigned. These constraints are achieved by providing a destructor for the sequence for this very purpose, as shown in Listing 23.5.

Listing 23.5 Constraints Enforced in the Destructor

```
Fibonacci_sequence::~Fibonacci_sequence() throw()
{
  using stlsoft::is_integral_type; // Using using declarations . . .
  using stlsoft::is_signed_type;   // . . . to fit in book. ;-)
  STLSOFT_STATIC_ASSERT(0 != is_integral_type<value_type>::value);
  STLSOFT_STATIC_ASSERT(0 == is_signed_type<value_type>::value);
}
```

You might think it strange to put in such constraints in a non-template class. The reason is simple: Maintenance programmers (including those who maintain their own code, hint, hint) are wont to change things without putting in all the big-picture research (i.e., reading all documentation). By putting in constraints, you are literally constraining any future changes from violating the design assumptions, or at least from doing so without extra thought.

Tip: Use constraints even in non-template classes to restrict and inform future maintenance activities.

I prefer to place constraints in the destructor of template classes because it's the method we can most rely on being instantiated. In non-template classes, I continue to use it for consistency.

23.3.5 Throw `std::overflow_error`?

One possible approach is to change the precondition enforcement assertion to be a legitimate runtime condition and to throw an exception. (The implementation shown in Listing 23.6 corresponds to *`Fibonacci_sequence_4.hpp`* on the CD.)

Listing 23.6 Version 4: Preincrement Operator

```
class_type& Fibonacci_sequence::const_iterator::operator ++()
{
  if(m_i1 < m_i0)
  {
    throw std::overflow_error("Exhausted integral type");
  }
  value_type  res   = m_i0 + m_i1;
  . . . // Same as Version 2
```

Although, in strict terms, this is a legitimate approach, it really doesn't appeal. The so-called exceptional condition is not an unpredictable emergent characteristic of the system at a particular state and time but an entirely predictable and logical consequence of the relationship between the

modeled concept and the type used to hold its values. Using an exception in this case just smacks of Java hackery.

I think it's clear at this point that we should either decide to represent the Fibonacci sequence as something that is genuinely infinite, with suitable indicators, or provide a mechanism for providing finite endpoints.

23.4 Discoverability Failure

Although the limit of a Fibonacci sequence for a given unsigned integral type is predictable and constant, requiring users of a type to know this either a priori or a posteriori is a bit rich, to say the least. Quite simply, people would not use such a component.

Our three current candidate implementations present unappealing alternatives.

1. Define the sequence without `end()`. This precludes any use of (`begin()`, `end()`) arguments to algorithms, but it does not preclude two iterators derived from `begin()` being used with algorithms. Further, there's nothing stopping users from gaily advancing their `begin()`-derived iterator past the point of overflow, and nothing to guide them in avoiding this.
2. Define the sequence with `end()` and rely on users' common sense not to use `end()` for anything at all. If they go into overflow, their program will die in a contract violation.
3. Throw an exception when overflow occurs. Despite this giving a tepid feeling of robustness, it's just as much a discoverability transgression as the other two options, and it also encourages a style of programming that is rightly confined to the world of virtual machines and seven-figure installation and deployment consultancy contracts.

Tip: Avoid using exceptions for failures that are a predictable result of the normal use of a component.

So, either the Fibonacci sequence is not something we should attempt to play with in an STL kind of way, or we need to apply some "finity" to it.

23.5 Defining Finite Bounds

There are two clear and related solutions to this problem.

1. Have `end()` return an iterator in the range [`begin()`, ∞) whose value does not overflow the value type.
2. Allow the user to specify an upper limit for the effective range provided by the sequence, represented in the value returned by `end()`. This value would have to be within the valid range.

A good implementation would provide both, where solution 1 is merely the default form of solution 2. We'll examine this by looking at the user-specified limits first.

23.5.1 Iterators Rule After All?

Before we proceed, I must cover an issue that some readers may now be considering. Earlier I ruled out the representation of Fibonacci sequences as independent iterators. The cunning linguists among you may be considering a form that does exactly that, as in:

```
std::copy(Fibonacci_iterator(), Fibonacci_iterator() + 40
    , std::ostream_iterator<Fibonacci_sequence::value_type>(std::cout
                                                            , " "));
```

In this case, the putative `Fibonacci_iterator` would implement the addition operator, such that the expression `Fibonacci_iterator() + 40` would evaluate to an instance that would terminate the iteration of a default-constructed iterator on its fortieth increment. At first blush this seems like an adequate solution to the problem.

However, the problem is that use of the addition operator on an iterator indicates that the iterator type is a random access iterator. Further, random access iterators have constant time complexity. To be sure, we're perforce violating pure STL requirements here and there in STL extension. But such violations are never done without due care and particular attention to the effects on discoverability and the *Principle of Least Surprise*. For example, it's hard to imagine that users of the **InetSTL** `findfile_sequence` class (Section 21.1), an STL collection that provides iteration of remote FTP host directory contents, will assume any particular complexity guarantees, given the vagaries of Internet retrieval. However, I suggest that it's far more likely that someone would assume constant time seeing pointer arithmetic syntax on an iterator.

Further, since a user will reasonably expect to be able to type `*(Fibonacci_iterator() + 40)` if he or she can type `Fibonacci_iterator() + 40`, we'd have to implement full random access semantics. But, as far as I know, there's no constant-time function integral formula with which you can determine the *N*th value of the Fibonacci sequence. (There are a couple of formulas that may be used, but they rely on the square root of five, which would rely on floating-point calculation. One of them is `((1 + sqrt(5)) / 2) - ((1 - sqrt(5)) / 2) ^ n`. I'm just enough of a computer numerist to know that I know far too little about floating point to be confident of writing a 100% correct sequence using floating-point calculations.)

Thus we would have to perform a number of forward or backward calculations to arrive at the required value, which is a linear-time operation. This would be a very unobvious violation of a user's expectations and is, in my opinion, unacceptable.

Tip: Beware of changing the complexity of built-in operators, particularly for random access iterators.

(Of course, we could provide amortized constant time by storing the calculated values in an array. We could go further and provide a static member array with precalculated values. We could even use template metaprogramming and effect compile-time calculation. But the purpose of this chapter is pedagogical. Feel free to do any of these, and let me know how it goes. I'll gladly post interesting solutions on the book's Web site.)

23.5.2 Constructor-Bound Range

One use case of a sequence might be to retrieve the first N elements in the sequence. It would be straightforward to implement the sequence and iterator classes such that you would specify the number of elements in the sequence constructor, which would then return a bounding iterator instance via its `end()` method, as shown in Listing 23.7. (This corresponds to *`Fibonacci_sequence_5.hpp`* on the CD.)

Listing 23.7 Version 5: Constructor and `end()` Method

```
public: // Construction
  explicit Fibonacci_sequence(size_t n) // Max # entries to enumerate
    : m_numEntries(n)
  {}
  . . .
public: // Iteration
  const_iterator begin();
  const_iterator end()
  {
    return const_iterator(m_numEntries); // Define end of sequence
  }
  . . .
```

You could use this as follows:

```
Fibonacci_sequence fs(25);

std::copy(fs.begin(), fs.end()
  , std::ostream_iterator<Fibonacci_sequence::value_type>(std::cout));
```

This could be implemented by adding an additional `m_stepIndex` member to `const_iterator`, which would be incremented each time `operator ++()` is called, and by evaluating equality (in `equal()`) by comparing the `m_stepIndex` members of the comparands (Listing 23.8).

Listing 23.8 Version 5: `const_iterator`

```
class Fibonacci_sequence::const_iterator
  : . . . // As shown previously
{
public: // Construction
  const_iterator(value_type i0, value_type i1)
    : m_i0(i0)
    , m_i1(i1)
    , m_stepIndex(0)
  {}
  const_iterator(size_t stepIndex)
    : m_i0(0)
    , m_i1(0)
```

```
    , m_stepIndex(stepIndex)
  {}
public: // Iteration
  class_type& operator ++()
  {
    . . . // Perform the advancement summations as before
    ++m_stepIndex;
    return *this;
  }
public: // Comparison
  bool equal(class_type const& rhs) const
  {
    return m_stepIndex == rhs.m_stepIndex;
  }
  . . .
```

This is a nice solution, and it also allows us to meaningfully support the `empty()` method. However, there's an equally valid use case, that of constraining the enumeration within a given integral range, for example, to enumerate all entries less than the value 1,000,000,000. The sequence might look like that shown in Listing 23.9. (This implementation corresponds to *`Fibonacci_sequence_6.hpp`* on the CD.)

Listing 23.9 Version 6: Constructor and `end()` Method

```
class Fibonacci_sequence
{
  . . .
public: // Construction
  explicit Fibonacci_sequence(value_type limit); // Value ceiling
    : m_limit(limit)
  {}
  . . .
public: // Iteration
  const_iterator begin();
  const_iterator end()
  {
    return const_iterator(m_limit); // Define sequence ceiling
  }
  . . .
```

Comparison would be conducted by the somewhat abstruse implementation of `equal()` shown in Listing 23.10. (There's an overflow bug in here, which I've left since this is a pedagogical class. Try setting the limit to 1,836,311,904 for a 32-bit unsigned value type. If readers want to implement the full testing for overflow, I'll be happy to post any correct solutions on the book's Web site.)

Listing 23.10 Version 6: `const_iterator::equal()` **Method**

```
bool
 Fibonacci_sequence::const_iterator::equal(class_type const& rhs)
  const
{
  if( 0 != m_i1 &&
      0 != rhs.m_i1)
  {
    // Both definitely normal iterable instances
    return m_i0 == rhs.m_i0 && m_i1 == rhs.m_i1;
  }
  else if(0 != m_threshold &&
          0 != rhs.m_threshold)
  {
    // Both definitely threshold sentinel instances
    return m_threshold == rhs.m_threshold;
  }
  else
  {
    // Heterogeneous mix of the two types
    if(0 == m_threshold)
    {
      return m_i0 >= rhs.m_threshold;
    }
    else
    {
      return rhs.m_i0 >= m_threshold;
    }
  }
}
```

A more flexible class would accommodate both these usage models. But doing so presents the sticky problem of how to unambiguously construct an instance of the sequence for either use case. One solution would be to use a two-parameter constructor, as follows:

```
  . . .
public: // Construction
  Fibonacci_sequence(size_t n, value_type limit);
  . . .
```

The parameter for the end-marker type not used would be given a stock value, for example:

```
Fibonacci_sequence(0, 10000); // This uses a limit of 10,000
Fibonacci_sequence(20, 0);    // This gives a sequence of 20 entries
```

Obviously, this is an inelegant and highly error-prone approach. A slightly less revolting alternative would be to use an enumeration to indicate the type of end marker required and use a `value_type` parameter for both the threshold and the number of entries:

```
  . . .
public: // Member Constants
  enum LimitType { thresholdLimit, countLimit };
public: // Construction
  Fibonacci_sequence(value_type limit, LimitType type);
  . . .
```

23.5.3 True Typedefs

The best solution uses true typedefs (Section 12.3), which facilitate the unambiguous overloading of essentially similar or even identical types. The final implementation of the `Fibonacci_sequence` does this, as shown in Listing 23.11. (This corresponds to *`Fibonacci_sequence_7.hpp`* on the CD.) Note the use of precondition enforcements in both constructors. A valid design alternative would be to throw `std::out_of_range` (since the user's value is not predictable).

Listing 23.11 Version 7: Class Declaration and Traits Class

```
template <typename T>
struct Fibonacci_traits;

template <>
struct Fibonacci_traits<uint32_t>
{
  static const uint32_t maxThreshold  = 2971215073;
  static const size_t   maxLimit      = 47;
};
template <>
struct Fibonacci_traits<uint64_t>
{
  static const uint64_t maxThreshold  = 12200160415121876738;
  static const size_t   maxLimit      = 93;
};

class Fibonacci_sequence
{
public: // Member Types
  typedef ?? uint32_t or uint64_t ??          value_type;
  typedef Fibonacci_traits<value_type>        traits_type;
  typedef true_typedef<size_t, unsigned>      limit;
  typedef true_typedef<value_type, signed>    threshold;
  class                                       const_iterator;
```

```
public: // Construction
  explicit Fibonacci_sequence(limit l = limit(traits_type::maxLimit))
    : m_limit(l.base_type_value())
    , m_threshold(0)
  {
    STLSOFT_MESSAGE_ASSERT( "Sequence limit exceeded"
                          , l <= traits_type::maxLimit());
  }
  explicit Fibonacci_sequence(threshold t)
    : m_limit(0)
    , m_threshold(t.base_type_value())
  {
    STLSOFT_MESSAGE_ASSERT( "Sequence threshold exceeded"
                          , t <= traits_type::maxThreshold());
  }
public: // Iteration
  const_iterator  begin() const;
  const_iterator  end() const
  {
    return (0 == m_limit)
              ? const_iterator(m_threshold)
              : const_iterator(m_limit, 0);
  }
public: // Size
  bool   empty() const
  {
    return 0 == m_limit && 0 == m_threshold;
  }
  size_t max_size() const
  {
    return traits_type::maxLimit;
  }
private: // Member Variables
  const size_t      m_limit;
  const value_type  m_threshold;
};
```

Note the use of the traits. Although they're not required by the definition of the sequence as it stands, they serve two important purposes. First, they provide a clear and obvious place for the limit and threshold magic numbers to reside, as well as making them largely self-documenting. Second, should you choose to use a 32- or 64-bit value type, the change involves just a single line.

The iterator class can now be defined as shown in Listing 23.12.

Listing 23.12 Version 7: `const_iterator`

```
class Fibonacci_sequence::const_iterator
  : . . . // As shown previously
{
public: // Member Types
  typedef Fibonacci_sequence::value_type      value_type;
  typedef Fibonacci_sequence::const_iterator  class_type;
private: // Construction
  friend class Fibonacci_sequence;
  const_iterator();
  const_iterator(Fibonacci_sequence::limit lim);
  const_iterator(Fibonacci_sequence::threshold t);
  . . . // Iteration and Comparison methods as before
};
```

With this definition, all the following are well defined (and thereby value constrained):

```
typedef Fibonacci_sequence  fibseq_t;
fibseq_t  fs(fibseq_t::limit(0));              // Empty sequence
fibseq_t  fs(fibseq_t::limit(1));              // 1 value
fibseq_t  fs(fibseq_t::limit(10));             // 10 values
fibseq_t  fs(fibseq_t::limit(47));             // 47 values
fibseq_t  fs;                                  // 47 values
fibseq_t  fs(fibseq_t::threshold(0));          // Empty sequence
fibseq_t  fs(fibseq_t::threshold(1));          // 1 value
fibseq_t  fs(fibseq_t::threshold(2));          // 3 values
fibseq_t  fs(fibseq_t::threshold(47));         // 10 values
fibseq_t  fs(fibseq_t::threshold(100));        // 12 values
fibseq_t  fs(fibseq_t::threshold(1000000000)); // 45 values
```

Equally important, the following are not well defined, and the user knows this because he or she can evaluate them against the member constants `Fibonacci_sequence::traits_type::maxLimit` and `Fibonacci_sequence::traits_type::maxThreshold`. Furthermore, because of the enforcements placed in the constructor bodies, the user finds out immediately when something is wrong, rather than at a later point during enumeration when the values overflow.

```
fibseq_t  fs(fibseq_t::limit(50));             // Breaks ctor precond
fibseq_t  fs(fibseq_t::threshold(2971215075)); // Breaks ctor precond
```

23.6 Summary

This chapter has covered the issues related to implementing an unbounded (infinite) notional collection. It has highlighted the imperfect fit between such collections and the strict finitude of C++'s integral types and the assumed boundedness of STL iterator pairs. Primarily, we've encountered, and eventually avoided, violations of the Goose Rule (Section 10.1.3).

We've seen that contracts—a mechanism for ensuring program adherence to design—are a thoroughly inappropriate mechanism for dealing with the conflict between conceptual infinity and the finitude of the language's types. We considered exceptions, but this was discounted because the exception point not only was not exceptional but also was entirely predictable. The requirement that users provide the limit to the range was unavoidable.

We determined two equally valid and desirable ways of limiting the range. Each was unambiguous, and providing a discoverable implementation was straightforward. But supporting both led to ambiguous and undiscoverable syntax, with unattractive compromises. Applying true typedefs saved the day, providing a class interface that is clear and discoverable, with the positive side effect that client code is itself more transparent.