Custom Iterators

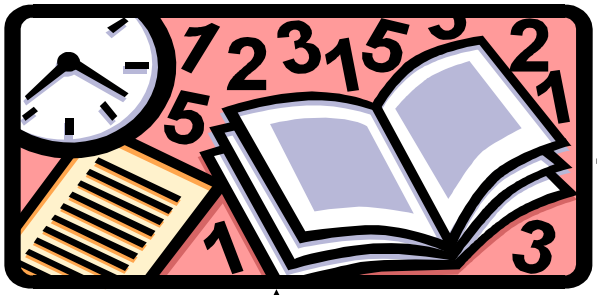
Christopher Baus and Thomas Becker
Zephyr Associates, Inc.
christopher@baus.net
thomas@styleadvisor.com

Financial Analytics

Number Cruncher



Chart Display



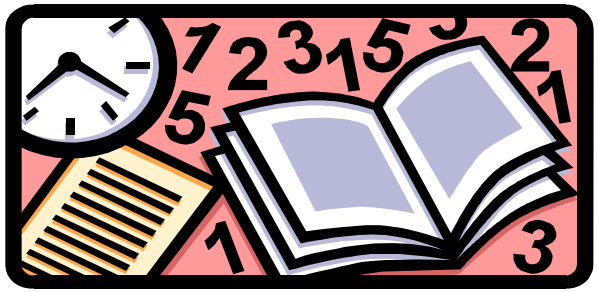




DB

Data Sources and Iterators

Number Cruncher



Exposes pairs of iterators

Stores sequences of numbers in STL containers

Chart Display

Data source: pairs of iterators



Time vs. Space Tradeoffs from Hell

Suppose the number cruncher has already stored two equally long sequences of numbers, and you need to expose to the client a third sequence representing the pointwise difference between these two sequences.

The naive way to do this is to store the sequence of differences in a third container and to expose iterators to this container.

Taking the difference between two doubles is an operation of almost negligible complexity.

Smart Iterators

```
class diff_it {
private:
  std::vector<double>::const_iterator it1;
  std::vector<double>::const_iterator it2;
public:
  double operator*() const
  {return *it1 – *it2;}

  diff_it operator++() const
  {++it_1; ++it_2; return *this;}
};
```

STL Container Iterators

Iterators exposed by STL containers are essentially pointers. They refer to a data item that exists in a certain memory location and is part of a range of data items. Moreover, they know how to get to the next item in that sequence, and possibly also the previous one.

The only possible difference between such an iterator and an actual pointer is that operations such as dereferencing or advancing may involve actions like retrieving the data within a node, or stepping from one node to the next.

STL Iterators

STL iterators are objects that expose a pointer-like interface. There is nothing in the STL specification of iterators that requires them to refer to data stored in memory.

STL iterators are essentially smart pointers.

Genericity

When writing an iterator such as the difference iterator that we were looking at before, try to find an appropriate level of genericity.

Instead of taking the difference of the numbers stored in two sequences, you may want to calculate the sum, or product, or Sharpe Ratio, or Treynor Ratio, …

Therefore, you should write a generic iterator that can do any of these things.



Generic Custom Iterators

Our paper presents several generic custom iterators with full source code, discussion, and examples of their usage.

Following is a brief overview of the iterators we present.

Combining Iterator

A combining iterator holds several iterators and a functional.

Moving the combining iterator is implemented as moving the member iterators in parallel.

Dereferencing the combining iterator is implemented as dereferencing the member iterators and applying the functional to the values thus obtained.

Processing Iterator

A processing iterator wraps a single iterator. The functionality that this wrapper adds to the plain iterator is calling a functional upon dereferencing.

The processing iterator is merely a special case of a combining iterator. Implementing it separately makes sense because the processing iterator has a simpler interface and some efficiency gains.

Generating Iterator

The combining iterator and the processing iterator focus on modifying and processing data that was obtained from one or more given iterators. The generating iterator, by contrast, creates its data “out of thin air” by just calling a functional that gets passed to the iterator upon construction.