The STL Model in the Geometric Domain

Ullrich Köthe[1] and Karsten Weihe[2]

[1] Fraunhofer–Institut for Computer Graphics Rostock
Joachim–Jungius–Str. 9, D-18059 Rostock, Germany
koethe@egd.igd.fhg.de
http://www.egd.igd.fhg.de/~ulli
[2] Universität Konstanz, Fakultät für Mathematik und Informatik
Fach D188, D-78457 Konstanz, Germany
weihe@fmi.uni–konstanz.de
http://www.fmi.uni–konstanz.de/~weihe

Abstract. Computational geometry and its close relative image analysis are among the most promising application domains of generic programming. This insight raises the question whether, and to which extent, the concepts of the *Standard Template Library* (*STL*) are appropriate for library design in this realm. We will discuss this question in view of selected fundamental algorithms and data structures.

1 Introduction

Throughout the last few years, *generic programming* has evolved as a new way of programming that is particularly well suited to algorithm–oriented software development. This is due to the specific role of algorithms and data structures in programs designed according to the ideas of generic programming. The conventional *structured–programming* approach puts emphasis on the algorithms, whereas data structures are treated as mere passive chunks of storage. On the other hand, *object–oriented programming* puts emphasis on data structures, and algorithms only appear as methods of data structures. In the latter approach, algorithms are nothing but "services" offered by data structures to access and modify the states of the objects.

In contrast, generic programming treats both algorithms and data structures as first–class citizens. Unlike object–oriented programming, the main design goal in generic programming is to *decouple* algorithms from data structures, and the main technique is to let data structures appear in a generic—and thus exchangeable—fashion inside algorithms. It is this loose coupling what makes the generic–programming approach promising for algorithm–oriented domains such as *computational geometry* and *computer vision*. Basically, we see three concrete reasons:

Complexity of design: Both data structures and algorithms are highly complex in these domains. Therefore, loose coupling between them is desirable, because separation makes the design much more manageable.

Maintainability: Experience shows that one of the main maintenance tasks in algorithmic software development is the adaptation of the underlying data structures to new, unforeseen requirements. If algorithms are strongly coupled to data structures, every such re–design also effects all algorithms. However, modifications of complex algorithms are expensive and highly error–prone and thus should be kept to a minimum.

Reusability: Many algorithms are potentially applicable to a wide range of data structures. However, if an algorithm is tightly coupled to a specific implementation of a specific data structure, this potential reusability cannot be realized.

At present, the *Standard Template Library* (*STL*, [6]) might be the best example for the application of the ideas of generic programming. Besides its specific purpose as a standard library of basic data structures and algorithms, the STL may be viewed as an "implemented repository" of detailed design concepts and techniques that are intended to support generic programming. When designing a generic library (in C++), it is natural to take patterns from the STL: the STL is the most mature fully generic library around, and its core has been integrated into the ANSI/ISO C++ standard. Hence, in the long run most C++ developers will become familiar with these concepts, and similar libraries will be easy to learn and use.

Familiarity with the STL will be very helpful for understanding the following discussion, see [6] for an introduction.

Iterators: The key concept of the STL for achieving loose coupling between data structures and algorithms is the *iterator*, which is among the basic design patterns in [1]. One of the fundamental innovations of the STL is to base the relationship between algorithms and data structures *exclusively* on iterators, i.e. algorithms do not see any data structures directly. This leads to a much more flexibile design because uniform iterators can be implemented for a wide range of data structures, even if the data structures themselves are very different.

The STL restricts itself to linear iterators. A valid iterator object i is associated with a *sequence* (array, linked list, file, etc.) and a specific position $p(i)$ in this sequence. An iterator allows access to the item at position $p(i)$ in the sequence (through method `operator*`), and can be moved to another position. At the very least, an iterator can traverse its sequence step–by–step, that is one position forward in every step. Depending on the underlying sequence type, an iterator could also offer methods for moving backwards (*e.g.* in doubly linked lists) or for random offsets (*e.g.* in arrays).[1]

From an algorithm's viewpoint, a sequence is given by a pair of iterators, i and j, which define a finite half–open range $[p(i), \ldots, p(j))$ of positions. Hence, traversing the items of a list amounts to moving iterator i forward until i equals j. The general pattern looks like this:

[1] This is reflected by the iterator categories of the STL.

```
template <typename Iterator>
void forEach(Iterator i, Iterator j) {
       for ( ; i != j; ++i )
          // do something with *i, the current item
}
```

In other words, i refers to the first position of the sequence and j to the position *one–past–the–end.* This is the natural view on many types of sequences (and thus will usually yield the most efficient implementations of iterators for these types). For instance, a one–past–the–end iterator j for a linked list might refer to the NULL item, and for a file it simply refers to the designated end–of–file position.

Geometry: This is one of the most promising application domains of generic programming. The question arises whether and to which extent the design concepts of the STL can—and should—be rigorously applied to the design of geometric algorithms and data structures. In this paper, we will try to give an answer to this question. First of all, we will see that the situation is much more complex than in the case of linear structures:

1. The items inside a container are not necessarily arranged in a linear fashion. For example, cyclic, multi–dimensional, and network–like data structures play an important role. These data structures offer non–linear access patterns, which in turn require new iterator categories.
2. In the geometric realm, data structures cannot always directly provide the iterators expected by a particular algorithm. There will be two kinds of mismatch:
   (a) The algorithm's navigation requirements may differ from what the data structure naturally provides. This requires some kind of adaptation.
   (b) The items of a container are usually assigned various *attributes.* Typically, a generic algorithm only operates on one or a few of them. However, it should be left open in the implementation of the algorithm which attributes are addressed (and how these attributes are implemented).
3. Many generic algorithms essentially realize non–trivial navigation patterns. It is favorable to interpret – and thus implement – such algorithms as an iterator rather than a sub-routine so that they can be passed to other algorithms.

We will analyse these aspects in view of various realistic use scenarios. Besides discussing new iterator categories, we will split the single level of indirection of the STL (the iterator) into several levels by introducing iterator adapters and so–called *data accessors.* Iterator adapters, although already present in the STL, will play a much more significant role in the geometric realm as a means of mapping different navigational patterns onto each other.

Data accessors [5] address Aspect 2(b). This concept allows one to strictly separate navigation over the items of a container from the access to item values. This means that an algorithm's view on a data structure is not formed exclusively

by iterators anymore, which gives us the ability to vary access functionality independently of the navigation patterns, as is needed when different algorithms (or different instantiations of the same generic algorithm) shall access different attributes of the same items in the same container.

2 Iterators for Non–Standard Cases

In this section, we discuss a few scenarios from the geometric realm, which deviate significantly from the STL–iteration model. It will turn out that the concept of half–open linear ranges is not really appropriate in these cases.

2.1 Cyclic Data Structures

Cyclic data structures play a prominent role in the geometric domain. Consider, for example, a polygon: a polygon is a sequence of points, connected by edges, where the last edge leads back to the first point. Thus, the polygon is a cyclic structure, and the choice of the first point is completely arbitrary. The same phenomenon can be observed with functions defined over an angle, such as sine and cosine, which are the basis for the most natural implementations of circles and ellipses. Again, the assignment of the angle 0 to a particular direction is completely arbitrary.

Although cyclic structures are still one-dimensional, we need an iterator concept beyond the STL model to describe them. This might seem surprising, but the reason for this lies in the lack of an explicit one–past–the–end position within a circular structure. Thus, the STL approach to specify a range by a pair of iterators does no longer work properly. This can best be seen by looking at the pair `[i, i)`. In the STL, this range uniquely specifies the empty range. However, for a cyclic structure a full revolution is the most likely interpretation, although it might just as well mean the empty range or even any number of consecutive revolutions. These different possibilities cannot be distinguished by the STL model.

One might try to solve the problem by introducing an auxiliary end element into the cycle. However, many algorithms operating on cycles want to choose the start/end positions arbitrarily, at runtime. In these cases a fixed end element would be counter-productive. Therefore, [2] introduced the *Circulator* as a new concept to capture periodic behavior. Like in the STL, we can define forward, bidirectional, and random access circulators, which basically provide the same set of functions as their STL counterparts. However, there are no limitations on how often a circulator can be incremented or decremented or on the admissible range of indexes for `operator[]`. Instead, we have the property that any circulator returns to itself after a number `n` of iterations (namely the number `n` of elements in the cyclic list):

```
Circulator i = ..., j=i;
int n = ...;
assert(advance(i, n), i == j);
```

All intermediate positions of the circulator refer to valid, referenceable items, no end-element exists.[2] To uniquely specify a range with circulators, an additional paramter is needed: the winding number. It tells an algorithm, how many full revolutions to do in addition to a range specified by a circulator pair. Thus, a complete circulator range is given by a triple `[begin, end, winding]`. For example, the triple `[i, i, 1]` corresponds to exactly one revolution, while the triple `[i, i, 0]` denotes the empty range. However, many algorithms only want to do exactly one revolution, in which case it is sufficient to pass a single iterator that refers to the starting position of the desired revolution. Then the basic iteration pattern looks like this:

```
template <typename Circulator>
void oneRevolution(Circulator begin)
{
    if(!begin.isSingular()) {
        Circulator i = begin;
        do {
            // do something with *i, the current item
        } while(++i != begin);
}
```

Example: Planar graphs are an illustrative example of circulators. A graph is defined as a set of *nodes* and a set of *edges*. Every edge is associated with two *incident* nodes, i.e. the edge connects these nodes. In turn, every node has a list of all its edges (the *incidence list*). A graph is called *planar* if it can be drawn in the plane such that no two edges cross each other (except at a common incident node). Figure 1 gives an example of a planar graph, and Figure 2 shows a graph for which it can be mathematically proved that no crossing–free drawing is possible. As can be seen, the incidence lists of planar graphs are cyclic data structures without a natural end.

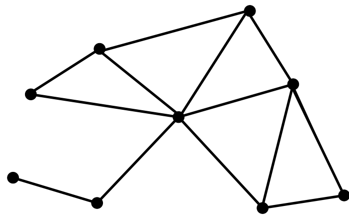

Fig. 1. A planar graph, embedded in the plane without crossings of edges.

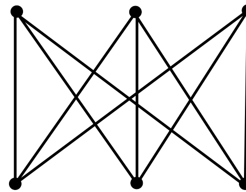

Fig. 2. A graph for which there is provably no embedding in the plane without crossings of edges.

[2] A singular circulator is introduced to denote the empty cycle. Its only valid operation is `i.isSingular()` which returns `true` iff the circulator is indeed singular.

Many algorithms on planar graphs are based on iteration over incidence lists as a basic operation. For example, many algorithms jump from node to node via edges. Whenever a new node is reached, its incident edges are scanned in clockwise or counterclockwise order. However, such a scan typically starts with the successor of the edge over which the node was entered. A designated begin and end of the list of incident edges does not make sense, since begin and end are only determined at runtime. Hence, the circulator model is more natural than the STL model.

2.2 Multi-dimensional Data Structures

The linear iteration style of the STL can be interpreted as navigationg in a 1-dimensional space. In computational geometry and computer vision we need to extend this design to higher dimensions [4]. For the sake of simplicity, we restrict the discussion to 2-dimensional data structures such as images and matrices, but the concepts discussed here can be applied to any dimension.

Since many algorithmS explicitly require access to the 2-dimensional structure, we need to define an appropriate 2-dimensional iterator concept. To do so, a new problem must be solved: we must be able to specify the coordinate direction a navigation command (such as incrementation or offset addition) refers to. In [4], it is proposed to represent each coordinate by a data member of the iterator like this:

```
class ImageIterator {
    ...

    typedef ... MoveX;
    typedef ... MoveY;

    MoveX x;    // refer to x-coordinate
    MoveY y;    // refer to y-coordinate
};
```

Navigation commands are sent to one of the two data members so that the desired direction is always clear.[3] Both `MoveX` and `MoveY` support all navigation commands of a standard (usually random access) STL iterator:

```
ImageIterator iimage = ...;

++iimage.x;    // advance one step to the right
++iimage.y;    // advance one step down

iimage.x -= 20;    // go 20 steps to the left
```

[3] For convenience, it is also possible to send navigation commands directly to the iterator via a 2-dimensional distance object `Distance2D`, which basically generalizes `ptrdiff_t` to 2-D. However, this does not raise fundamental new issues and will not be discussed here (cf. [4] for details).

It should be noted that `MoveX` and `MoveY` are not themselves iterators since they do not support access to the actual data items, i.e. we cannot write `*iimage.x` or `*iimage.y`. Only the entire iterator has complete knowledge of the current position, so we always have to write `*iimage`.

Another important difference to the STL is found when we start to think about how to mark the borders of iteration. A linear sequence has only 2 ends, which can be represented by two iterators. In contrast, an image of size $w \times h$ has $2(w+h+2)$ past-the-border positions.[4] It is not very realistic to pass iterators for all of these positions on every call of an algorithm. Yet, a complete representation of the image's boundary is necessary, because many algorithms cannot foresee where a given iterator will pass the border. Consider, for example, an algorithm that controls drawing on an image with the mouse. When the mouse leaves the image's range at an arbitrary, unforseeable location, the algorithm must recognize it and stop drawing ("clipping").

The solution is to calculate new border marker as needed from a few markers at known positions. Of course, this requires that navigation commands are still applicable to past-the-border iterators which don't refer to valid pixels. For example, in an algorithm performing a row–major iteration, we need to mark the end of the first column and the end of the current row. If we advance the iterator to the next row, the marker at the end of the row needs to be advanced as well. To initialize this algorithm, we need at least three iterators: one to the upper left corner of the image, and two to the end of the first row and column respectively. However, if we use random access iterators, we can improve upon this by realizing that the two end iterators can be calculated from just one marking the position beyond the lower right corner of the image. Even simpler, we can than directly compare the `x` and `y` members of the two iterators. This results in the following basic navigation pattern:

```
template <typename ImageIterator>
void forEachPixel(ImageIterator upperleft, ImageIterator lowerright) {
    for(; upperleft.y != lowerright.y; ++upperleft.y) {
        ImageIterator i = upperleft;
        for(; i.x != lowerright.x; ++i.x) {
              // do something with *i, the current item
        }
    }
}
```

Figure 3 illustrates our convention of specifying a rectangular region by passing a pair of iterators to the upper left and beyond the lower right corners of the region. That is, lower right is outside the region, similarly to the end iterator in the STL.

Note that even in case of the `forEach()` algorithm it is necessary to use a 2-dimensional iterator. If we tried to use a linear iterator (such as a scan-order

[4] Past-the-border positions are those outside locations that have at least one direct or diagonal neighbor in the image.

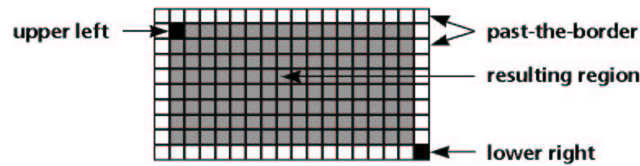

Fig. 3. Two image iterators at upper left and lower right (black squares) determine the region of interest (gray squares and upper left). There are 2(w+h+2) past-the-border locations (white squares and lower right).

iterator, which returns all pixels in row-major order), we would get incorrect subrange semantics. In image processing, a subrange usually means a subimage, which we can easily specify by moving upper left and lower right inwards by the appropriate amount. In contrast, a subrange of the scan-order iterator would correspond to a number of consecutive lines plus two incomplete lines above and below them, which is hardly what we want.

The necessity of 2-dimensional access becomes even more apparent, if we consider algorithms that involve a neighborhood around the current pixel. As a simple example, consider an algorithm that replaces a specific pixel by the average grey value of its 3×3 neighborhood (for simplicity, we omit bounds checking):

```
void avaragePixel(ImageIterator current)
{
    ImageIterator upperleft = current - Distance2D(1, 1);
    ImageIterator lowerright = current + Distance2D(1, 1);
    int sum = 0;

    for(; upperleft.y != lowerright.y; ++upperleft.y) {
        ImageIterator i = upperleft;
        for(; i.x != lowerright.x; ++i.x) sum += *i
    }
    *current = sum / 9;  // the average
}
```

Since images and matrices are usually random access data structures, one may ask why we are not simply using indexing instead of iterators. There are several reasons why iterators are more flexible:

- As in the STL, the iterator solution includes the indexing solution: a two dimensional indexing operator is supported by the `ImageIterator` via the syntax `i(10, 20)`.
- Iterators are easier to use with offsets relative to an arbitrary location. Instead of doing index calculations by hand, an algorithm can simply place the iterator on the reference position and use the relative indexes directly.

– It is easier to write iterator adapters if we already have iterators in the first place. Several interesting iterator adapters for the ImageIterator will be discussed in section 3.2.

2.3 Algorithms as Iterators

In our attempt to decompose functionality into orthogonal concepts, we realized that many geometric algorithms can be devided into a navigation part and a part that does not involve navigation (i.e. a part, that does some computations with the current data items). Both types of functionality vary essentially independently of each other. Look, for example, at graph traversal algorithm: many different algorithms use the same fundamental depth-first or breadth-first traversal patterns. When we encapsulate these patterns into independent building blocks, we can reuse one implementation with all these algorithms. When we implement them in addition in form of iterators rather than subroutines, it becomes much easier to pass these building blocks around, and we can even pass them to algorithms that don't know anything about graphs. For example, by passing a depth-first or breadth-first iterator to an algorithm like `print()`, we can print node information in either order without the need to implement a new algorithm.

The graph traversal example induces an interesting insight: iterators implementing algorithms do not always support the STL model of half-open ranges, because this requires that the end is known in advance. Often, the end is found only in the course of traversal, so that it must be reported via an `isEnd()` member function of the iterator. this method returns `true` if and only if all items of the sequence have been passed. We will call an iterator supporting the `isEnd()` function a *loose-end iterator.*

The most prominent example for loose-end behavior is an input stream: its end is only indicated by an EOF flag, which is set after the stream was closed. The STL (in its `istream_iterator`) chooses not to introduce a new categorie for this behavior, but the experiences within the geometric realm suggest, that this phenomenon might be so common as to justify a separate category. In section 3.3 we show, how the loose-end model can be adapted to the STL model.

Graph Traversal There are various strategies for exploring a (connected) graph, for example *depth–first search* and *breadth–first search.* Such a strategy visits the nodes one after another, but the iteration is implemented by a navigation over the incidence structure of the graph (by jumping from node to node along edges).

It is quite natural to implement these strategies as a node iterator. Among other advantages, this makes it possible to visit all nodes once without the need for an additional list in which all nodes are maintained redundantly. The design of such an iterator class should incorporate the following insights:

1. It does not make much sense to specify the nodes to be visited as a range $[i \ldots j)$. In fact, only the start node i is known before the algorithm is executed, whereas $i+1, \ldots, j-1$ are unknown until the end of the algorithm.

2. In particular, the notion of subranges is not supported as it is for linear containers.
   There is one notable exception: it makes perfect sense to restrict such a strategy to a "subrange" that forms a prefix of the sequence of nodes visited regularly by the strategy. For example, depth–first seach is often used to generate a path from some node s to another node t, which can be specified by a *closed* range $[s \ldots t]$. A specification through a half–open range in STL style is not possible.
3. However, other kinds of prefixes cannot be specified by any kind of ranges (neither closed nor half–open nor open). For instance, breadth–first search proceeds in layers: first all nodes connected to the start node are visited (first layer), then all unvisited nodes connected to the first layer (second layer), and so on. Sometimes one is interested in iterating over all nodes in the layers $1 \ldots k$ for some given k. Again, this forms a prefix, however, neither the last node of the prefix nor the first node of the rest is known, and thus no delimiter for the set of nodes to be visited can be specified.

A graph–search iterator class is easily and naturally implemented as a loose–end iterator as described above. In contrast, a one–past–the–end iterator does not make sense, because there is no designated one–past–the–end position—neither for the whole range of nodes nor for typical definitions of subranges.

3 Iterator Adapters

In the last section, we have identified a few scenarios in which a one–past–the–end iterator would be awkward or even meaningless. Nonetheless, it is often desirable to apply STL–style algorithms to such iterators. What we need here are *iterator adapters*, which adapt the syntax of their adaptees to the STL style. We will consider the individual scenarios from Sextion 2 in the same order in the following subsections.

3.1 From Cyclic Structures to the STL Model

In Section 2.1 we have seen that circular data structures are naturally described by circulators. The direct use of circulators requires special algorithms which know about the circulator's properties. This is o.k. as long as these algorithms are indeed most naturally implemented in terms of circulators, such as many algorithms on planar graphs (see Section 2.1). However, in some cases we will need to apply algorithms expecting an STL–compatible iterator to a cyclic data structure.

This can easily be achieved by wrapping circulators into STL–conforming adapters which encapsulate the special knowledge about circulator properties. In particular, these adapters hide the additional winding number, which says how many full revolutions the iterator is supposed to do. Two adapters compare equal if they point to the same location and their winding numbers are equal. This could be implemented as follows:

```
template <typename Circulator>
class CirculatorAdapter
{
    Circulator begin_, current_;
    int winding_number_;
  public:
    CirculatorAdapter(Circulator c, int winding_number)
    : begin(c), current_(c), winding_number_(winding_number)
    {}

    CirculatorAdapter<Circulator> & operator++() {
        ++current_;
        if(current_ == begin_)  ++winding_number_;
        return *this;
    }

    bool operator==(CirculatorAdapter<Circulator> const & c) const {
        return current_ == c.current_ &&
               winding_number_ == c.winding_number_;
    }
    ...
};
```

Given a circulator at an arbitrary position, it is now straightforward to construct an STL–conforming begin/end pair for exactly one circulation. In this case, begin and end just differ by the initialization of the winding number:

```
Circulator c = ...;
CirculatorAdapter<Circulator> begin(c, 0), end(c, 1);
```

Other ranges can be constructed similarly. Of course, we pay for compatibility with a slight runtime overhead (some additional checks), but in many applications this can be tolerated, considering the advantages.

3.2 One–Dimensional Subsets Of Multi–Dimensional Structures

Many algorithms operating on images are not interested in the entire image, but only in a one-dimensional subset of the pixels. There are many possibilities to define such subsets, and many algorithms don't care which subset is actually used as long as it is one-dimensional. Consequently, we implement these algorithms in terms of a linear iterator or circulator, and select the appropriate subset independently. Examples for useful linear subsets include single rows and columns, arbitrary straight lines, splines and even space filling curves such as Hilbert's curve. We can also define many different circular structures such as circles, ellipses, rectangles, and the contours of arbitrary regions. These subsets are best implemented by means of adapters, mapping the two-dimensional image iterator to linear iterators or circulators respectively. Except under extremely tight performance constraints, it doesn't make sense to implement these one-dimensional

views directly on top of the raw matrix, because this would require a separate implementation for each data structure, whereas the adapters can be reused for anyanything providing an `ImageIterator`.

To give an idea of the possibilities, we will describe an iterator adapter moving along an arbitrary straight line in the image. This `LineIterator` implements Bresenham's algorithm for traversing a line. That is, given a start point and an offset, the iterator will calculate the necessary increments and will adavance by one pixel in the desired direction upon every call of `operator++`. A simple implementation of this adapter could look like this:

```
template <typename ImageIterator>
class LineIterator
{
    ImageIterator current_;
    float dx_, dy_, x_, y_;

  public:
    LineIterator(ImageIterator i, int xoffset, int yoffset)
    : current_(i), x_(0), y_(0), dx_(0), dy_(0)
    {
        int number_of_steps = max(abs(xoffset), abs(yoffset));
        if(number_of_steps > 0) {
            dx_ = (float)xoffset / number_of_steps;
            dy_ = (float)yoffset / number_of_steps;
            x_ = dx_ / 2.0;
            y_ = dy_ / 2.0;
        }
    }

    LineIterator & operator++()
    {
        x_ += dx_;
        if(x_ >= 1.0) {
            x_ -= 1.0;
            ++current_.x;
        }
        else if(x_ <= -1.0) {
            x_ += 1.0;
            --current_.x;
        }
        // likewise for the y coordinate
    }
    ...
};
```

Now we have several possibilities to mark the end of iteration. The most natural one is probably to implement the iterator as a loose–end iterator, because it knows its end anyway (note the variable `number_of_steps` in the constructor). This would correspond to the observation that many iterators implementing algo-

rithms support the loose-end model (cf. Section 2.3). Alternatively, we can choose to provide iterator pairs. For example, we might create an end iterator by passing an `ImageIterator` at the line's target position to the `LineIterator`'s constructor. The working iterator compares equal to this iterator, when its `current_` member has reached the target position. However, this does not conform to the STL model, because we got a *closed* rather than a half–open interval (an at-the-end rather than one–past-the-end iterator). Additional effort is required to really construct a one–past-the-end iterator, which might indicate that the STL model is not the most appropriate one in this case.

The big advantage of these geometric iterators is that the number of algorithms can be reduced. For example, we only need one drawing algorithm for any kind of shape, because the shape is solely determined by the iterator. This pays off as soon as we have several algorithms that operate on different kinds of shapes.

3.3 Adaptation of Loose–End Iterators

Recall the loose–end iterator model from Section 2.3. A general template for adapting iterators with an `isEnd`–method to the STL style could look like this (brief sketch):

```
template <typename Iterator>
class IteratorAdapter
{
    Iterator iter_;
    bool is_dummy_;

  public:
    IteratorAdapter() : is_dummy_(true) {}

    IteratorAdapter(Iterator i)
    : iter_(i), is_dummy_(false) {}

    bool operator==(IteratorAdapter<Iterator> const & o) const {
        return (iter_.isEnd() || is_dummy_ ) ?
                            (o.iter_.isEnd() || o.is_dummy) :
                            (!o.is_dummy && iter_ == o.iter);
    }
    ...
};
```

4 Attribute Access

In [5], we proposed an extension of the STL guidelines, which we named *data accessors*. Basically, an object of a data–accessor class encapsulates the access to one specific attribute. It takes an iterator and a data accessor to read and write the value of an attribute for an item. For example, consider a sequence

of type `struct Point`, where `Point` contains the `double` members `x` and `y`. Reading and writing attribute `x` for the item referred to by STL–style iterator `iter` is expressed by `(*iter).x`. This reveals the implementation of the abstract attribute `x` to all algorithms. In contrast, a data accessor `da` comes with methods `get` and `set` for reading and writing this value:

- Reading: `cout << da.get (it);`
- Writing: `da.set (it, 1.0);`

For example, when data accessors are incorporated into the design, the STL function `replace` would be declared like this:

```
template <typename ForwardIterator, typename DataAccessor,
          typename T>
void replace (ForwardIterator begin, ForwardIterator end,
              DataAccessor da, T old_value, T new_value);
```

To replace all x coordinates equal to 1.0 by 2.0 in a sequence `seq` of struct type `Point`, we could use the template class `MemberAccessor`, which was implemented in [5] and internally keeps a pointer–to–member [7]:

```
template <typename Str, typename T>
class MemberAccessor
  {
  public:
    MemberAccessor (T Str::*ptr)
         : i_ptr(ptr) { }
    template <typename Iterator>
       T get (Iterator it) const
          { return (*it).*i_ptr; }
    template <typename Iterator>
       void set (Iterator it, T value)
          { (*it).*i_ptr = value; }
  private:
    T Str::*i_ptr;
  };
```

This class can be applied as follows:

```
MemberAccessor<Point,double>  da (&Point::x);
replace (begin, end, da, 1.0, 2.0);
```

We refer to [5] for a more detailed and systematic discussion on the technical level[5] and to [8] for a discussion on the conceptual level in view of another

[5] This includes a critical discussion of another, more “STL–like” approach, which allows one to apply the STL function `replace` to a single item attribute instead of the item data as a whole: implementing an adapter class for iterators, whose sole purpose is to overwrite `operator*` such that it returns the attribute instead of the whole struct.

algorithmic domain: graph algorithms. In the next subsection, we will focus on another aspect, which has not been explicitly addressed in any previous work on data accessors but is at the heart of generic programming: to implement each component on the highest possible level of generality.

Remark : This concept even makes it possible to select the coordinate (*i.e.* x or y) at run time, namely when the object `da` is instantiated. This aspect has been systematically investigated in [3].

4.1 Flexible Combination of Iterators and Data Accessors

Consider an algorithm such as the STL algorithm `remove_if`, which applies some predicate to all items of a sequence. In the STL design of `remove_if`, the predicate receives a value of the item type of the sequence for evaluation. The predicate is applied to an item by applying `operator*` of an iterator referring to this item and calling the evaluation method of the predicate with the result as the (unique) argument. However, the iterator object itself potentially yields more information than the result of `operator*`, and sometimes it is desirable to base the predicate on this additional information. We will first give an example where this additional information is necessary. Afterwards, we will show that data accessors allow generic implementations of such predicates.

For the sake of argument, suppose we are given a list of points as above, and we want to implement a predicate for items of this list that evaluates to `true` if the item is not the very last one and the x coordinate of this item is smaller than the x coordinate of the very next item. Hence, this predicate needs the values of both the current and the very next item. However, this predicate shall be implemented for algorithms such as `remove_if`, which do not "know" that the predicate accesses more than just the current position.

This example shows the limits of the STL version of `remove_if`: in many cases, it does not suffice to hand over the current item's value to the predicate. In fact, the predicate needs the current position to retrieve the next item's value. Of course, this position is given by an iterator.

However, it is desirable to implement such a predicate class as generically as possible. This class should cover all cases in which the value of some attribute of the current item is compared with the value of the same attribute for the very next item. So the following aspects should be left open in the implementation of the predicate class:

1. the comparison operation,
2. the type of the attribute,
3. the *name* of the attribute as a member of the struct,
4. even the fact that this attribute is simply implemented as a member of the (struct) item type.

Here is an implementation of this predicate class based on data accessors:

```
template <typename Iterator, typename Accessor, typename Comparator>
class SuccessorComparator
{
  Iterator    end_;
  Accessor    acc;
  Comparator  cmp_;

public:
  SuccessorComparator (Iterator end, Accessor acc, Comparator cmp)
  : end_(end), acc_(acc), cmp_(cmp)
  {}

  template <typename Iterator>
  bool operator() (Iterator it)
  {
    return it != end && cmp_ (acc_.get(it), acc_.get(it+1));
  }
};
```

Obviously, this implementation leaves open the above–declared aspects. On the other hand, leaving open the last two aspects might be next to impossible if data accessors (or an equivalent concept) are *not* used.

Remark : Again, a loose–end iterator would be more natural here: the first operand to the "`&&`" inside `operator()` would reduce to a call `it.isEnd()`, and there were no need for an additional end iterator object. This would allow us to make `Iterator` a template argument of `operator()` instead of `SuccessorComparator` itself, which means that one object of `SuccessorComparator` could serve all iterator types that are defined on the point list. In fact, `SuccessorComparator` itself only depends on `Iterator` because of the internal end iterator.

Conclusion

The STL was designed in view of fundamental, all–purposes algorithms and data structures. In this paper, we have seen that a specific application domain may suggest design decisions that are incompatible with the STL design. The question arises whether the design of a domain library should obey the STL guidelines or the needs of the domain. In the long run, the STL guidelines will be part of every C++ developer's background knowledge. This is a strong pragmatic argument in favor of the STL guidelines.

On the other hand, we hope that not only the syntax, but also the *spirit* of the STL will find its way into every C++ developer's mind. In our opinion, the spirit of the STL suggests focusing on the needs of the domain:

1. First focus on what the algorithms actually need from the data structures.
2. Then design the interface between data structures and algorithms according to these requirements.

3. Finally, if the interface offered by a data structure does not match the interface required by an algorithm, implement adapters to get the syntax right.

In our case this means: design your domain–specific library according to the domain–specific needs (as it was done in Section 2). If the STL algorithms are potentially useful for your data structures, provide appropriate adapters (like in Section 3).

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.L.: Design patterns, elements of reusable object–oriented software Addison–Wesley, 1994.
2. Fabri, A., Giezeman, G.-J., Kettner, L., Schirra, S., Schönherr, S.: On the Design of CGAL, the Computational Geometry Algorithms Library. ETH Zürich, Department Informatik, Technical Report no. 291, 1998
3. Gluche, D., Kühl, D., Weihe, K.: Iterators evaluate table queries. ACM Sigplan Notices **33**(1):22–29 (1/1998).
4. Köthe, U.: Reusable Software in Computer Vision. to appear in: B. Jaehne, H. Haussecker, P. Geissler (eds.): Handbook of Computer Vision and Applications, Vol. 3, Academic Press 1998
5. Kühl, D., Weihe, K.: Data access templates. C++ Report **9**(7):15–21 (7/1997)
6. Musser, D., Saini, A.: STL Tutorial and Reference Guide. Addison–Wesley, 1995.
7. Stroustrup, B.: The C++ Programming Language (third edition). Addison–Wesley, 1997.
8. Weihe, K.: Reuse of algorithms — still a challenge to object–oriented programming. Proceedings of the 12th ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications (OOPSLA), 34–48 (1997).
9. Weihe, K.: A software engineering perspective on algorithmics. http://www.informatik.uni–konstanz.de/~weihe/manuscripts.html#paper35 (1998)