6 Reusable Software in Computer Vision

s:uk

Ullrich Köthe

Fraunhofer-Institut für Graphische Datenverarbeitung
Rostock, Germany

6.1 Introduction

s:uk.intro

Since the mid 1980ies general purpose computers such as workstations and PCs have reached performance levels that make them increasingly interesting for computer vision applications. As compared with applications based on special hardware, software solutions have a number of important advantages: they are easily transferred to the next hardware generation, can be adapted quickly to new requirements, and can be reused in many different contexts and systems.

It is, however, not easy to reach these goals in practice. The current praxis of software development in computer vision has not lead to software that is as flexible and reusable as we would like it to be. I believe that these problems are due to two fundamental difficulties, which will be addressed in this chapter:

- **The Flexibility vs. Performance Problem**
  In many cases, flexibility introduces a run-time overhead which degrades performance. Given the huge amount of data which needs to be processed in most computer vision applications, there is only a very limited leeway to trade performance for flexibility. Thus, we must look for flexibility techniques which don't affect performance.
- **The Framework Coupling Problem**
  Most reusable components are currently provided as parts of large frameworks. Since individual components in these frameworks are tightly coupled together, it is not possible to take out just those pieces we actually need: the framework only functions as a whole. This essentially prevents the use of components from several frameworks in the same project. Thus, we must provide reusable software in form of independent building blocks that can be reused individually.

Let us illustrate these problems with an example. Suppose we have built an image processing application which we would like to adapt to the special needs of an important customer. The existing body of functionality is large, so that we can not afford to throw it away and start over from scratch. To fill in the missing pieces quickly, we would like to reuse some functionality that has been developed in another context, perhaps another project or the public domain.

For example, suppose we want to incorporate the Shen-Castan edge detector that is part of the KHOROS system [2] into our environment. Unfortunately, this algorithm only runs on a specific variety of the KHOROS image format. If we do not happen to use this image format in our system, we have an adaptation problem. This problem is traditionally solved in several ways:

- We can convert our original image representation into the required KHOROS format prior to every application of the algorithm, and convert the results back upon completion. This is the solution applied within KHOROS itself and within the *component-based* software paradigm in general (cf. Sect. 4). Clearly, conversion is relatively inefficient, because it takes time and memory. Thus, we would rather like to avoid it at the lower level of a system (although it might be a good option at higher levels).
- To minimize conversion, we can opt to build an entire module using KHOROS functionality. I.e., besides the edge detector we also use KHOROS functions for the necessary pre- and post-processing. Although it might work in some cases, it is not a good idea to use several different frameworks within the same project — it leads to explosive growth in code size, and requires to understand and maintain all these different frameworks which might easily turn down the positive effect of software reuse.
- If we have access to the source code (which is the case with KHOROS functionality), we can also attempt to modify the code as to adapt it to the needs of our environment (e.g., let it run on top of our image format). This approach is very problematic since it easily introduces subtle bugs that are very hard to detect. To avoid mistakes, a very thorough understanding of the existing code is required, which many people find even harder than re-implementing the solution from scratch.

Neither of these options is really satisfying. It would be much better if reusable components were provided as *independent building blocks* which do not depend on a specific environment (such as KHOROS) and can be adapted without knowledge of their inner workings (in particular, without source code modification) and without loss of performance. The design methods described in previous chapters (Chaps. 4 and 5) do not support independent building blocks very well, especially at the level of basic algorithms and data structures, where performance is at a premium. Therefore, in this chapter we describe an alternative design approach called "generic programming" which provides very good support for definition and flexible integration of independent building blocks. Generic programming complements the other methods so that we can design combined solutions (such as object-oriented frameworks delegating their implementation to generic components) which emphasize the strengths of each method.

The most important dependency we need to remove is the dependency of algorithms upon specific data representations. To a certain extend, object-oriented programming provides a solution to this problem if data is accessed via abstract classes and virtual functions. An

abstract image base class could, for example, look like this (all code examples are in C++):

```
class AbstractImage
{
  public:
    virtual unsigned char getPixel(int x, int y) const = 0;
    virtual void setPixel(unsigned char value, int x, int y) = 0;
    virtual int width() const = 0;
    virtual int height() const = 0;
    // ...
};
```

Algorithms using this abstract base class can not know what's behind the virtual functions. Thus, we can wrap arbitrary image formats into subclasses of this abstract image, for example the KHOROS xvimage format:

```
class KhorosImage
: public virtual AbstractImage
{
  public:
    virtual unsigned char getPixel(int x, int y) const
    {
        return image->imagedata[x + y*image->row_size];
    }
    virtual void setPixel(unsigned char v, int x, int y)
    {
        image->imagedata[x + image->row_size*y] = v;
    }
    virtual int width() const { return image->row_size; }
    virtual int height() const { return image->col_size; }
    // ...
  private:
    xvimage * image;
};
```

However, there are two fundamental problems with this approach. First, the virtual function call introduces a run-time overhead. In Sect. 6.6 we report benchmark results that show a performance penalty of up to 500 % when we use virtual functions instead of accessing the data directly via pointers[1]. Of course, the overhead could be avoided if we implemented algorithms as member functions of the concrete image classes, giving them direct access to the data. But this would counter our desire for reusability, since these member functions would once again depend on the concrete image format and were not independently reusable.

The second, more severe problem is the return type of the function `getPixel()`. In a statically typed language, this return type must be fixed (to `'unsigned char'` in the example) so that `'float'` and RGB images can not be subclasses of `AbstractImage`. This could be overcome by introducing another abstract class `AbstractPixel` but this would require even more virtual calls so that it is not a practical pos-

[1] The overhead results primarily from the inability of the compiler to inline virtual functions. See Chapter 5 for further discussions.

sibility. Thus, for different return types of `getPixel()` we still need different versions of each algorithm.

Although organizing objects into class hierarchies has advantages (see Chap. 5), in a statically typed language it also introduces static dependencies. i. e., all types (base classes, function arguments, and data members) and member functions that are mentioned in a class must also be transferred into the new environment if we want to reuse this class. This often leads to a chain of interdependencies which is the main cause for the framework coupling problem mentioned above, and prevents the definition of really *independent* building blocks.

Therefore, an additional design technique is needed. Traditionally, the problems mentioned have been addressed by *code generators*. Code generators produce the source code for a concrete application from an abstract implementation of a building block and a description of the context (containing, among other things, the concrete data types involved). Meanwhile, these code generating facilities have been built directly into major object-oriented languages (such as C++, ADA, and Eiffel) under the name of *genericity* and *parametric types*. The most extensive support for genericity is provided by the *template mechanism* in C++. The widespread support of genericity enabled the development of a new programming paradigm — *generic programming* — which generalizes and formalizes the ideas of code generation and thus solves the above mentioned problems without introducing a major run-time overhead. In this chapter we will describe, how generic programming can be used to design reusable, independent building blocks for computer vision applications.

6.2 Generic programming

s:uk.genprog

Generic programming has been introduced as an independent programming paradigm by Musser and Stepanov [6, 7]. It became popular when it was chosen as the underlying paradigm of the C++ Standard Library, in particular the part of the library which is known as the *Standard Template Library* [5]. It is especially suited to implement reusable data structures and algorithms at the lower levels of a system. Since generic programming is still relatively new, we will give a short general introduction before we start applying it to computer vision.

6.2.1 Generic data structures

s:uk.genprog.datastruct

In the first step of generic programming, container data structures are made independent of the type of the contained objects. In computer vision, we will implement *generic image data structures* that can contain arbitrary pixel types like this:

```
template <typename PIXELTYPE>
class GenericImage
{
  public:
    PIXELTYPE getPixel(int x, int y) const {
        return data_[x + width()*y];
    }

    void setPixel(PIXELTYPE value, int x, int y) {
        data_[x + width()*y] = value;
    }

    int width() const { return width_; }
    int height() const { return height_; }
    // etc.

  private:
    PIXELTYPE * data_;
    int width_, height_;
};
```

Alternatively, we can also use an additional array of pointers to the start of each line to enable double indirection which is faster on many machines:

```
template <typename PIXELTYPE>
class GenericImageAlt
{
  public:
    PIXELTYPE getPixel(int x, int y) const {
        return lines_[y][x];
    }

    // etc.

  private:
    PIXELTYPE *  data_;
    PIXELTYPE ** lines_;
    int width_, height_;
};
```

In either case, the generic type `PIXELTYPE` acts as a placeholder for any pixel type we might want to store, and the corresponding image data structure will be automatically generated by the compiler when needed. For example, to create an RGB image, we could implement an RGB compound, and instantiate a `GenericImage` with it:

```
template <typename COMPONENTTYPE>
struct RGBStruct
{
    RGBStruct(COMPONENTTYPE r, COMPONENTTYPE g, COMPONENTTYPE b)
    : red(r), green(g), blue(b)
    {}

    COMPONENTTYPE red, green, blue;
};

// generate an new image data type holding RGBStruct<unsigned char>
typedef GenericImage<RGBStruct<unsigned char> > RGBStructImage;
```

6.2.2 Generic algorithms

s:uk.genprog.algo

The basic techniques to define generic data structures are well known and already widely applied. In the next step of generic programming we define algorithms as templates as well. For example, to copy arbitrary images (including the `RGBStructImage` defined above) we could write:

```
template <typename SRCIMAGE, typename DESTIMAGE>
void copyImage(SRCIMAGE const & src, DESTIMAGE & dest)
{
    for(int y=0; y<src.height(); ++y)
      for(int x=0; x<src.width(); ++x)
        dest.setPixel(src.getPixel(x, y), x, y);
}
```

This algorithm can be applied to any image pair for which a conversion from the source to the destination pixel type exists, provided the source image supports operations `src.height()`, `src.width()`, and `src.getPixel(x,y)`, and the destination image `dest.setPixel(v,x,y)`. We will call requirements like these an algorithm's *required interface* since they reflect what the user of the algorithm must ensure to allow the compiler to generate the code for a given pair of images.

Two properties of generic algorithms should be noted here: first, instantiation of the algorithm is completely type based. i. e., any image that fulfills the required interface can be used within the algorithm — no inheritance from a specific abstract base class is required. Second, all decisions regarding flexibility are done at compile time (so called *compile-time polymorphism*). Thus, when all access functions are expanded inline, the compiler can generate code which runs as fast as a traditional pointer-based (non-reusable) implementation[2].

6.2.3 Iterators

s:uk.genprog.iter

In practice, it turned out that algorithms with required interfaces similar to the previous one (i. e., algorithms which directly access the data structures' member functions) are not the optimal solution. Instead, generic programming introduces dedicated *interface objects* which explicitly decouple algorithms and data structures from each other. For our present purpose, the most important interface objects are *iterators* [1]. Iterators encapsulate how to navigate on a container data structure, i.e., they point to a specific element (pixels) within a specific container and can be moved to refer to others.

In the Standard Template Library (STL), Musser and Stepanov define five iterator categories which completely cover the needs of algorithms

[2] In practice, many compilers will generate slightly slower code for generic algorithms because current optimizers are more familiar with pointers than with genericity. There is, however, no fundamental reason that generic algorithms must be slower, and some compilers (such as KAI C++) can already generate equally fast code in both cases.

operating on top of linear (one-dimensional) data structures. These iterator categories have been developed on the basis of a thorough analysis of what different algorithms need to do their tasks, and what different data structures can efficiently implement. For our present discussion, three categories are of importance:

Forward Iterators allow to go forward to the next element (`++iterator`), to find out if two iterators point to the same position (`iterator1 == iterator2`) and to access the data at the current position (`*iterator`). This functionality is sufficient to implement the `copy()` algorithm.

Bidirectional Iterators also provide a function to go backwards to the previous element (–iterator). This functionality would be needed to implement a `reverseCopy()` function.

Random-Access Iterators additionally allow to jump back and forth by arbitrary offsets (`iterator += n, iterator -= n`), to compare iterators according to an ordering (`iterator1 < iterator2`), and to access an element at a given distance of the current position (`iterator[n]`). This functionality is required by most sorting algorithms, e.g. `quicksort()`.

Typically, a singly linked list provides forward, and a doubly linked list bidirectional iterators, while a vector supports random-access iteration. It can be seen that the iterators adopt the syntax of C pointers, which always conform to the requirements of random-access iterators. Using iterators, the copy function for linear (one-dimensional) data structures will look like this:

```
template <typename SrcIterator, typename DestIterator>
void copy(SrcIterator s, SrcIterator end, DestIterator j)
{
    for(; s != end; ++s, ++d)   *d = *s;
}
```

By convention, the iterator `end` marks the end of the iteration by pointing to the first position *past the sequence*. In Section 6.3, we will generalize these iterator categories to two-dimensions, i.e., images.

6.2.4 Functors

s:uk.genprog.functors

To improve the flexibility of algorithms, Stepanov and Musser have introduced *functors*. Functors encapsulate an important sub-computation which we want to vary independently of the enclosing algorithm. We can exchange the respective sub-computations by simply passing different functors to the algorithm. For illustration, consider the `transform()` algorithm. We can interpret `transform()` as a generalization of `copy()`: while `copy()` writes the source data into the destination sequence unchanged, `transform()` performs an arbitrary transformation in between. Since there are infinitely many possible transformations we en-

capsulate them into a functor so that a single `transform()` template can be used for all possible transformations:

```
template <typename SrcIterator, typename DestIterator, typename Functor>
void transform(SrcIterator s, SrcIterator end
               DestIterator d, Functor func)
{                                  // ^^^^ pass functor
    for(; s != end; ++s, ++d)   *d = func(*s);
}                                    // ^^^^ apply functor
```

A functor behaves like a function pointer, but it is both more powerful (since it can store internal state) and more efficient (since the compiler can inline it). We will see more examples for functors when we implement the basic image processing functionality in Sect. 6.5.2.

s:uk.genprog.accessor

6.2.5 Data accessors

The idioms reported so far (generic data structures and algorithms, iterators, and functors) are the basis of the STL. However, experience has shown that algorithms and data structures are not sufficiently independent as long as we have only a single function

`iterator::operator*()`

to access the iterator's current data item. There are two fundamental problems with this function: First, it assumes that all algorithms want to see the entire data at the current position. This makes it problematic to operate on a subset of the actual data structure (such as a single color band of an RGB image). Second, `*iterator` must be used for both reading and writing the data. This means, `operator*()` must return the data by reference if we want to overwrite it. This is difficult to implement for certain container implementations (e. g., a multiband RGB image).

Therefore, additional interface objects, called *data accessors*, are needed to encapsulate actual data access (Kühl and Weihe [3]). A data accessor reads data at the current iterator position via a `get()` function, and writes them via a `set()` function. Within these functions, arbitrary adaptations (such as extracting just the red band of an RGB pixel, of transforming RGB to gray) can be performed. In general, a data accessor looks like this:

```
template <typename VALUETYPE, typename ITERATOR>
struct SomeAccessor
{
    typedef VALUETYPE value_type;  // advertise your value type

    value_type get(ITERATOR & i) const;   // read an item
    void set(value_type v, ITERATOR & i) const;  // write an item
};
```

Using this data accessor, we can implement a more general version of the linear `copy()` algorithm like this:

```
template <typename SrcIterator, typename SrcAccessor,
          typename DestIterator, typename SrcAccessor>
void copy(SrcIterator s, SrcIterator end, SrcAccessor sa,
          DestIterator d, DestAccessor da)
{
    for(; s != end; ++s, ++d)   da.set(sa.get(s), d); // read/write via accessor
}
```

Although it is probably hard to see the advantages of accessors (and generic programming in general) in an algorithm as simple as `copy()`, the same techniques can be applied to arbitrary complicated algorithms, where reuse will really pay off.

s:uk.genprog.traits

6.2.6 Meta-information: Traits

To automate instantiation of more complicated generic algorithms, we also need meta-information, i.e. machine readable descriptions of the properties and constraints of generic building blocks. In C++, meta-information is commonly strored in so called *traits* classes Myers [8]. Let's illustrate traits using two important examples.

As is well known, C++ performs certain automatic *type conversions*, if an expression can not be evaluated otherwise. For example, integers are promoted to double if they are mixed with real numbers in an arithmetic expression. In case of generic algorithms, we can not know in advance which type conversions should be applied. Traits can be used to provide the meta-information needed in such a situation. We define `PromoteTraits` which describe the type promotion rules between two types `T1` and `T2` as follows:

```
template <typename T1, typename T2> struct PromoteTraits;

struct PromoteTraits<float, unsigned char> {
    typedef float Promote;   // unsigned char is promoted to float
};

struct PromoteTraits<RGBStruct<float>, RGBStruct<unsigned char> > {
   typedef RGBStruct<float> Promote; // RGBStruct<unsigned char> is
};                                   // promoted to RGBStruct<float>
```

These traits classes must be defined for all type combinations we are interested in. Then we can use RGBStruct-traits to implement a generic function for, say, component-wise addition of two `RGBStructs`:

```
template <typename T1, typename T2>
PromoteTraits<RGBStruct<T1>, RGBStruct<T2> >::Promote
operator+(RGBStruct<T1> const & t1, RGBStruct<T2> const & t2)
{
    // construct result from promoted sums of color components
    return PromoteTraits<RGBStruct<T1>, RGBStruct<T2> >::Promote(
          t1.red + t2.red, t1.green + t2.green, t1.blue + t2.blue);
}
```

The traits class determines the type of the result of the addition for any combination of `RGBStruct<T1>` and `RGBStruct<T2>` for which we have defined `PromoteTraits`. In addition to binary promotion traits,

we also define unary traits that are used to determine the type of temporary variables for intermediate results (including the initialization of these variables). These definitions look like this:

```
template <typename T> struct NumericTraits;

struct NumericTraits<unsigned char> {
    // store intermediate results as float
    typedef float Promote;

    // neutral element of addition
    static unsigned char zero() { return 0; }
    // neutral element of multiplication
    static unsigned char one() { return 1; }
};

struct NumericTraits<RGBStruct<unsigned char> > {
    typedef RGBStruct<float> Promote;

    static RGBStruct<unsigned char> zero() {
        return RGBStruct<unsigned char>(0, 0, 0);
    }
    static RGBStruct<unsigned char> one() {
        return RGBStruct<unsigned char>(1, 1, 1);
    }
};
```

The necessity of using `NumericTraits` is best seen in a function calculating the average of some values. Look at the following traditional code:

```
unsigned char value[20] = {...};

float average = 0.0;
for(int i=0; i<3, ++i)
          average += value[i];
average /= 20.0;
```

Had we stored the average in the original data type `unsigned char`, a numeric overflow would probably have occurred after a few additions, leading to a wrong result. We will use `NumericTraits` to avoid this problem. The generic implementation of the averaging algorithm on a one-dimensional sequence will look like this:

```
template <typename Iterator, typename Accessor>
NumericTraits<Accessor::value_type>::Promote
average(Iterator i, Iterator end, Accessor a)
{
    NumericTraits<Accessor::value_type>::Promote sum =
                     NumericTraits<Accessor::value_type>::zero();

    int count = 0;
    for(; i != end; ++i, ++count)  sum = sum + a.get(i);

    return (1.0 / count)  * sum;
}
```

6.3 Two-dimensional iterators

s:uk.2diter

6.3.1 Navigation

s:uk.2diter.navig

Although linear iterators as described in the previous section have their place in computer vision (see Sect. 6.7), we will first and foremost work on two-dimensional images. Thus, we must generalize the iterator concept to two dimensions (higher dimensions can be introduced analogously, but will not be covered in this chapter). i. e., we must define navigation functions that tell the iterator in which coordinate direction to move. As a first idea, we could define navigation functions like `incX()` or `subtractY()`, but in C++ there exists a more elegant solution that uses nested classes to define different views onto the same navigation data. More specifically, consider the iterator design:

```
class ImageIterator {
  public:
    // ...

    class MoveX {
        // data necessary to navigate in X direction
      public:
        // navigation function applies to X-coordinate
        void operator++();
        // ...
    };

    class MoveY {
        // data necessary to navigate in Y direction
      public:
        // navigation function applies to Y-coordinate
        void operator++();
        // ...
    };

    MoveX x;  // x-view to navigation data
    MoveY y;  // y-view to navigation data
};
```

Now we can use the nested objects x and y to specify the direction to move along as if we had two one-dimensional iterators:

```
ImageIterator i;
++i.x;  // move in x direction
++i.y;  // move in y direction
```

This way we can define 2-D iterators in terms of the same operations which the STL defines for 1-D iterators. Table 6.1 lists the complete functionality to be supported by 2-dimensional random access iterators[3]. A standard implementation that works with all image formats which internally store image data as a linear memory block is contained on the CD-ROM coming with this book.

[3] Forward and bi-directional image iterators can easily be defined by dropping some of the listed functions.

Table 6.1: *Requirements for image iterators. Meaning of symbols: ImageIterator i, j; int dx, dy; struct Dist2D { int width, height; } dist;*

Operation	Result	Semantics
ImageIterator::MoveX		type of the x navigator
ImageIterator::MoveY		type of the y navigator
ImageIterator::value_type		type of the pixels
`++i.x; i.x++`	`void`	increment x coordinate[1,3]
`--i.x; i.x--`	`void`	decrement x coordinate[1,3]
`i.x += dx`	`ImageIterator::MoveX&`	add dx to x coordinate
`i.x -= dx`	`ImageIterator::MoveX&`	subtract dx from x coord.
`i.x - j.x`	`int`	difference of x coordinates[2]
`i.x = j.x`	`ImageIterator::MoveX&`	`i.x += j.x - i.x`[2]
`i.x == j.x`	`bool`	`j.x-i.x == 0`[1,2]
`i.x < j.x`	`bool`	`j.x-i.x > 0`[2]
`++i.y; i.y++`	`void`	increment y coordinate[3]
`--i.y; i.y--`	`void`	decrement y coordinate[3]
`i.y += dy`	`ImageIterator::MoveY&`	add dy to y coordinate
`i.y -= dy`	`ImageIterator::MoveY&`	subtract dy from y coord.
`i.y - j.y`	`int`	difference of y coordinates[2]
`i.y = j.y`	`ImageIterator::MoveY&`	`i.y += j.y - i.y`[2]
`i.y == j.y`	`bool`	`j.y-i.y == 0`[2]
`i.y < j.y`	`bool`	`j.y-i.y > 0`[2]
`ImageIterator k(i)`		copy constructor
`k = i`	`ImageIterator&`	copy assignment
`i += dist`	`ImageIterator&`	add offset to x and y
`i -= dist`	`ImageIterator&`	subtract offset from x and y
`i + dist`	`ImageIterator`	add offset to x and y
`i - dist`	`ImageIterator`	subtract offset from x and y
`i - j`	`Dist2D`	difference in x and y[2]
`i == j`	`bool`	`i.x==j.x && i.y==j.y`[2]
`*i`	`value_type &`	access the pixel data[4]
`*i`	`value_type`	read the pixel data[5]
`i(dx, dy)`	`value_type &`	access data at offset (dx, dy)[4,6]
`i(dx, dy)`	`value_type`	read data at offset (dx, dy)[5,6]

[1] prefer this in inner loops
[2] i and j must refer to the same image
[3] `*i.x++, *i.y++` etc. not allowed
[4] mutable iterator only, optional
[5] constant iterator only, optional
[6] replaces operator[] in 2-D

t:uk.imiter

All functions listed must execute in constant time. To further optimize execution speed, we also label some functions as "prefer this in inner loops" which means that these functions should be fastest (i. e., have the smallest "big O factor"). Typically, they should consist of just one addition respective comparison to match the speed of normal pointer arithmetic. All navigation functions must commute with each other, i. e., the same location must be reached if a sequence of navigation functions (say `++i.x; ++i.y;`) is applied in a different order (like `++i.y; ++i.x;`).

s:uk.2diter.daccess

6.3.2 Direct data access

In contrast to the STL, the image iterator functions for direct data access (`operator*` and `operator()`) have been made *optional*, because there is no guarantee that these functions can be defined as desired when we want to implement iterators for *existing data structures* which we can not change. For example, suppose for some reason we must use the `AbstractImage` defined in the introduction. To make it compatible with our generic algorithms, we need to wrap it into an image iterator. This is actually very easy, since we can use plain integers for the iterators' `MoveX` and `MoveY` members:

```
struct AbstractImageIterator
{
    typedef unsigned char value_type;
    typedef int           MoveX;
    typedef int           MoveY;
    int x, y;

    // init iterator at the upper left corner of the image
    AbstractImageIterator(AbstractImage * img)
    : image_(img), x(0), y(0)
    {}

    // point two iterators to different positions ?
    bool operator!=(AbstractImageIterator const & i) const
    {
        return (x != i.x || y != i.y);
    }
    // ... not all functions shown

    // delegate data access to the underlying image
    unsigned char get() const
    {
        return image_->getPixel(x, y);
    }

    void set(unsigned char v)
    {
        image_->setPixel(v, x, y);
    }

    /* these functions are hard or impossible to implement:
    unsigned char & operator*();
    unsigned char & operator()(int dx, int dy);
    */
```

```
  private:
    AbstractImage * image_;
};
```

For a mutable iterator, `operator*` and `operator()` are required to return a reference to the current data item. However, since the underlying `AbstractImage` does not return data by reference, the iterator can't do it either (Tricks to work around this problem, such as the proxy technique described in Meyers [4], provide partial solutions at best). Therefore, in Table 6.1 these functions are marked "optional". A general and satisfying solution to this problem is a major motivation behind the introduction of *data accessors* in Sect. 6.4.

6.3.3 Image boundaries

s:uk.2diter.imbound

In contrast to a 1-D sequence, which has just 2 ends, an image of size $w \times h$ has $2(w+h+2)$ past-the-border pixels[4]. To completely mark the boundaries of this image, we need a set of iterators — one for the begin and end of each row and each column. Iterators that refer to past-the-border pixels will be called *boundary markers* (no new type is required). According to their position we distinguish *left*, *right*, *top*, and *bottom* boundary markers. It is not desirable to pass a set of $2(w+h+2)$ boundary markers to an algorithm. Therefore we specify rules how the algorithm can create any boundary marker from just a few on known positions.

All 2-D iterators must be able to navigate on past-the-border as if these pixels were inside the image. Thus we can transform an iterator into a boundary marker and a boundary marker into another one. The difference is that boundary markers must not be dereferenced, i.e. we are not allowed to access data at past-the-border positions. To be exact, top boundary markers are defined as follows:

- For each column a unique top boundary marker exists. A program may hold many identical instances of this marker.
- Applying `++i.y` will move the boundary marker to the first pixel of the current column, a subsequent `--i.y` recreates the marker. Likewise, `i.y+=dy` may be applied if the target pixel is inside the image or past the bottom border ($dy \le h+1$). Subsequent `i.y-=dy` recreates the marker.
- Applying `++i.x`, `--i.x`, `i.x+=dx` or `i.x-=dx` produces the top boundary markers of the respective target columns, if they exist.

Analogous definitions apply to the other boundaries. Examples for generic algorithms implemented on the basis of these definitions are given below.

[4] Past-the-border pixels are the outside pixels having at least one 8-neighbor in the image.

6.4 Image data accessors

s:uk.ida

When we implemented an image iterator for the `AbstractImage` we saw that it was impossible to provide data access operators (`operator*` and `operator()`) that returned the current data item by reference. Another common example for a data structure suffering from this problem is the *multiband RGB image*. In a multiband RGB image, pixel values are stored in three separate images (bands) for each of the three color components, rather than in one image of compound RGBStructs (as, for example, the `RGBStructImage` defined in section "Generic programming"). A typical implementation of a *multiband RGB image* could look like this:

```
class MultibandRGBImage
{
  public:
    unsigned char & red(int x, int y);
    unsigned char & green(int x, int y);
    unsigned char & blue(int x, int y);
    //...
  private:
    unsigned char * redband, * greenband, * blueband;
};
```

The corresponding image iterator would simply mirror the access functions of the image:

```
struct MultibandRGBImageIterator
{
    // navigation functions not shown ...

    unsigned char & red();
    unsigned char & green();
    unsigned char & blue();

    /* this is difficult or impossible to implement
    RGBStruct<unsigned char> & operator*();
    RGBStruct<unsigned char> & operator()(int dx, int dy);
    */
};
```

Once again we can not implement the standard access functions used within the STL, because the desired return type

```
RGBStruct<unsigned char>&
```

does not physically exist in the underlying image data structure. Thus, while it is easy to define a uniform navigation interface for the iterators, we can not guarantee that all iterators have a uniform data access interface. Kühl and Weihe [3] proposed an additional level of indirection, the *data accessor*, to recover the desired uniform interface. As was mentioned earlier, data accessors provide a pair of `get()` and `set()` functions which are inlined by the compiler so that the additional indirection does not affect performance. In cases where the iterator provides `operator*`, get and set simply call it:

```
template <typename VALUETYPE, typename STANDARDITERATOR>
struct StandardAccessor
{
    typedef VALUETYPE value_type;  // advertise your value type

    value_type get(STANDARDITERATOR & i) const {
        return *i;   // read current item
    }

    void set(value_type v, STANDARDITERATOR & i) const {
        *i = v;      // write current item
};
```

Since the `AbstractImageIterator` does not work this way, it needs a different accessor that could look like this:

```
struct AbstractImageAccessor
{
    typedef unsigned char value_type;

    value_type get(AbstractImageIterator & i) const {
        return i.get();
    }
    void set(value_type v, AbstractImageIterator & i) const {
        i.set(v);
    }
};
```

In addition to the standard `get()` and `set()` functions, an RGB accessor provides functions to access each color separately. These functions will be used by algorithms which explicitly require RGB pixels. For the multiband RGB image, such an accessor could be defined as follows:

```
struct MultibandRGBImageAccessor
{
    typedef RGBStruct<unsigned char> value_type;
    typedef unsigned char component_type;

    value_type get(MultibandRGBImageIterator & i) const {
        return value_type(i.red(), i.green(), i.blue());
    }

    void set(value_type rgb, MultibandRGBImageIterator & i) const {
        i.red() = v.red;      // assume that the iterator
        i.reen() = v.green;   // mirrors the red(), green(),
        i.blue() = v.blue;    // blue() function of the image
    }                          // which return data by reference

    component_type getRed(MultibandRGBImageIterator & i) const {
        return i.red();
    }

    void setRed(component_type v, MultibandRGBImageIterator & i) const {
        i.red() = v;
    }
    // ...
};
```

Providing a uniform data access interface regardless of the underlying image implementation is not the only advantage of introducing accessors: they can also be used to perform an arbitrary transforma-

tion before the algorithm actually sees the data (for instance, color to gray conversion), or to restrict access to just a part of the current item, such as one color component of an RGB pixel. For example, we could select just the red band of the RGBStructImage by using the following accessor:

```
struct RedComponentAccessor
{
    typedef unsigned char value_type;

    value_type get(RGBStructImage::Iterator & i) {
        return (*i).red;
    }

    void set(value_type v, RGBStructImage::Iterator & i) const {
        (*i).red = v;
    }
};
```

For example, we could now apply a scalar convolution to each color component separately by calling the convolution algorithm (Sect. 6.5.3) with red, green, and blue band accessors in turn. Without accessors, this behavior would be difficult to implement for the RGBStructImage, whereas it would be easy for a MultibandRGBImage. Hence accessors serve to smooth out the differences between various implementation choices for the image data structures which would otherwise make universally reusable algorithm implementations impossible.

6.5 Generic algorithms on images

s:uk.genericalg

6.5.1 Basic image processing algorithms

s:uk.genericalg.basic

Now that we have covered all preliminaries we can finally come to the real thing, the implementation of generic algorithms on images. Basic image processing algorithms will be direct generalizations of the one dimensional algorithms described in section “Generic Programming”. The region of interest will be determined by a pair of iterators, called upperleft and lowerright. Similarly to the end iterator in 1-D, which points to the first position *past* the desired range, lowerright denotes the first pixel *outside* the desired ROI (i. e., one pixel right and below of the last pixel inside the ROI), while upperleft points to the first pixel in the ROI (see Fig. 6.1). Note, that upperleft and lowerright may refer to arbitrary positions in the image, so that any algorithm can be applied to any rectangular subregion of an image without modification.

The basic image processing functionality can be implemented with just five generic functions:

initImage()	inspectImage()	inspectTwoImages()
copyImage()	transformImage()	combineTwoImages()

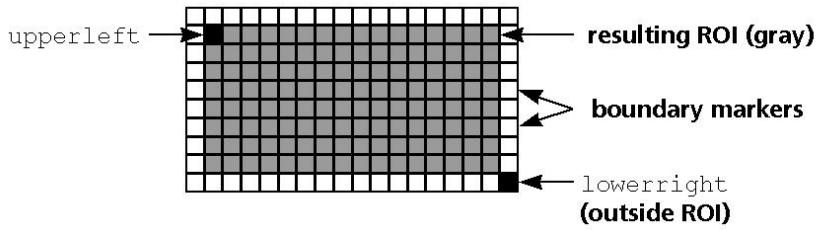

Figure 6.1: *Iterators at `upperleft` and `lowerright` determine the region of interest*

f:uk.genericalg.ROI

With the exception of `copyImage()`, all these functions take functors to define their actual semantics (see next section). The first is used to load specific, possibly location dependent, values in a new image. The next two are applied to collect statistics for the entire image resp. labeled regions. As the name suggests, `copyImage()` is used to copy images (which may require an automatic type conversion). Since we can pass iterators at arbitrary positions, this also covers the functionality of extracting and inserting subimages. Finally, the last two functions will be used to perform point operations involving one image (such as thresholding or histogram manipulation) and two images (such as pixel arithmetic) respectively. To give an idea, how these functions work, we will give the implementation of two of them:

```
template <class SrcIterator, class SrcAccessor, class Functor>
void inspectImage(SrcIterator upperleft, SrcIterator lowerright,
                  SrcAccessor sa, Functor & func)
{
    // iterate down first column
    for(; upperleft.y < lowerright.y; ++upperleft.y)
    {
        SrcIterator s(upperleft);   // s points to begin of row
        for(; s.x < loweright.x; ++s.x)   // across current row
                        func(sa.get(s));    // apply functor
    }
}

template <class SrcIterator, class SrcAccessor,
          class DestIterator, class DestAccessor, class Functor>
void transformImage(
    SrcIterator supperleft, SrcIterator lowerright, SrcAccessor sa,
    DestIterator dupperleft, DestAccessor da, Functor func)
{
    for(; supperleft.y < lowerright.y;      // down first column
                        ++supperleft.y, ++dupperleft.y)
    {
        SrcIterator s(supperleft);  // s and d point to begin of
        DestIterator d(dupperleft); // current src resp. dest row
        for(; s.x < loweright.x; ++s.x, ++d.x)     // accross row
        {
            da.set(func(sa.get(s)), d);         // apply functor
        }
    }
}
```

To extend the functionality towards arbitrary *regions of interest* (ROIs) we also introduce versions of these algorithms that use mask images. Application of these algorithms' functionality is restricted to the positions where the mask returns true. These algorithms are denoted by the suffix `'If'`. `copyImageIf()` looks like this:

```
template <class SrcIterator, class SrcAccessor,
          class MaskIterator, class MaskAccessor,
          class DestIterator, class DestAccessor>
void copyImageIf(
    SrcIterator supperleft, SrcIterator lowerright, SrcAccessor sa,
    MaskIterator mupperleft, MaskAccessor ma,
    DestIterator dupperleft, DestAccessor da)
{
    for(; supperleft.y < lowerright.y;
            ++supperleft.y, ++dupperleft.y, ++mupperleft.y)
    {
        SrcIterator s(supperleft);
        MaskIterator m(mupperleft);
        DestIterator d(dupperleft);
        for(; s.x < loweright.x; ++s.x, ++d.x, ++m.x)
        {
            if(ma.get(m))              // check the mask
                da.set(sa.get(s), d);  // copy only if true
        }
    }
}
```

In conjunction with various functors (see next section), these functions are extremely versatile. By passing different iterators and accessors, we can apply them to

- arbitrary pixel data types (`byte`, `int`, `float`, `RGBStruct` or anything else),
- arbitrary image data structures (`GenericImage`, `AbstractImage`, `MultibandRGBImage` or anything else),
- subsets of structured pixel types (E.g., by passing a `RedBandAccessor` and a `GreenBandAccessor` to `copyImage()`, we can copy the red band of an RGB image into the green band of another.),
- debug versions of the data structures (E.g., we can use a bounds-checking iterator during debugging.),
- arbitrary subimages (by passing iterators at suitable positions and/or use mask images).

Neither of these options requires any changes to the algorithms. Thus, we need much less source code than a traditional library to provide the same functionality. In our generic library VIGRA, the algorithms described above, plus functors for all common pixel transformations (such as thresholding, algebraic functions, pixel arithmetic, and logical functions), take less than 10 Kbytes of source code. In contrast, in a traditional system like KHOROS, all these functions must be implemented repeatedly which consumes over 100 Kbytes of source

code. These numbers directly translate into a huge gain in programmer productivity.

s:uk.genericalg.functors

6.5.2 Functors for image processing

To bring the above functions to life, we need to implement functors for common operations. For example, consider an algorithm that maps an image from an arbitrary scalar intensity domain into the displayable range 0...255. To do this we first need to find the minimal and maximal pixel values in the original image. This can be done by the function `inspectImage()` and the following `FindMinMaxFunctor`[5]:

```
template <class PIXELTYPE, class ITERATOR>
struct FindMinMaxFunctor
{
    MinmaxFunctor() : count(0) {}

    void operator()(ITERATOR & i)
    {
        if(count == 0) {
            max = min = *i;  // first pixel: init max and min
        }
        else {
            if(max < *i) max = *i;   // update max
            if(*i < min) min = *i;   // and min
        }
        ++count;   // update pixel count
    }

    int count;
    PIXELTYPE min, max;
};

Image img(width, height);
//...
FindMinMaxFunctor<Image::PixelType, Image::Iterator> minmax;

inspectImage(img.upperLeft(), img.lowerRight(), img.accessor(),
             minmax);
```

Now that we know the minimum and maximum, we can apply a linear transform to map the original domain onto the range 0...255. We use the function `transformImage()` in conjunction with a functor `LinearIntensityTransform`:

```
template <class PIXELTYPE, class ITERATOR>
struct LinearIntensityTransform
{
    typedef typename NumericTraits<PIXELTYPE>::Promote value_type;

    LinearIntensityTransform(value_type scale, value_type offset)
    : scale_(scale), offset_(offset)
    {}
```

[5]For convenience, in the sequel we will assume that we are using an image data type that has member functions `upperLeft()`, `lowerRight()`, and `accesor()`, as well as embedded `typedefs PixelType, Iterator` and `Accessor`. Also, the iterator shall support `operator*`.

```
    value_type operator()(ITERATOR & i) const
    {
        return scale_ * (*i + offset_);
    }

    value_type scale_, offset_;
};

//...

LinearIntensityTransform<Image::PixelType, Image::Iterator>
         map_intensity(255.0*(minmax.max-minmax.min), -minmax.min);

Image destimg(width, height);

transformImage(img.upperLeft(), img.lowerRight(), img.accessor(),
           destimg.upperLeft(), destimg.accessor(), map_intensity);
```

At first this looks rather complicated and lengthy. One should, however, keep in mind that the functions and functors need to be implemented only once, and can then be reused for any image we want to transform, regardless of the pixel types and implementation details. Moreover, some further simplifications to the above code are possible. For example, we can define factory functions to generate the triples `[upperleft, lowerright, accessor]` resp. pairs `[upperleft, accessor]` automatically, given an image. By using expression templates, we can also let the compiler generate functors automatically, given the expression it implements. However, a description of these simplifications is beyond the scope of this chapter.

6.5.3 Higher level functions

s:uk.genericalg.advanced

The techniques outlined above are likewise applied to implement higher level algorithms. Currently, we have implemented about 50 generic algorithms in our library VIGRA. This includes edge and corner detection, region growing, connected components labeling, feature characterization, and calculation of feature adjacency graphs. A few examples shall illustrate some possibilities.

First lets look at a simple averaging algorithm. We will give its full implementation to illustrate typical image iterator usage in operations involving a window around the current location. This implementation was also used as the generic variant in the benchmark tests reported in the next section[6].

```
template <class SrcImageIterator, class SrcAccessor,
          class DestImageIterator, class DestAccessor>
void averagingFilter(SrcImageIterator supperleft,
                     SrcImageIterator slowerright, SrcAccessor sa,
                     DestImageIterator dupperleft, DestAccessor da,
                     int window_radius)
{
```

[6] Of course, in practice one would use a separable algorithm to gain speed. Section 6.7.1 shows how this is effectively done using row and column iterator adapters.

```
    // determine pixel type of source image
    typedef typename SrcAccessor::value_type SrcType;

    // determine temporary type for averaging  (use
    // NumericTraits::Promote to prevent overflow)
    typedef typename NumericTraits<SrcType>::Promote SumType;

    int width = slowerright.x - supperleft.x;  // calculate size
    int height = slowerright.y - supperleft.y; // of image

    for(int y=0; y < height;
                      ++y, ++supperleft.y, ++dupperleft.y)
    {
        SrcImageIterator xs(supperleft);   // create iterators
        DestImageIterator xd(dupperleft);  // for current row

        for(int x=0; x < width; ++x, ++xs.x, ++xd.x)
        {
            // clip the window
            int x0 = min(x, window_radius);
            int y0 = min(y, window_radius);
            int winwidth  = min(width - x, window_radius + 1) + x0;
            int winheight = min(height - y, window_radius +1) + y0;

            // init sum to zero
            SumType sum = NumericTraiTs<SrcType>::zero();

            // create y iterators for the clipped window
            SrcImageIterator yw(xs - Dist2D(x0, y0)),
                             ywend(yw + Dist2D(0, winheight));

            for(; yw.y != ywend.y; ++yw.y)
            {
                // create x iterators for the clipped window
                SrcImageIterator xw(yw),
                                 xwend(xw + Dist2D(winwidth, 0));

                // fastest operations in inner loop
                for(; xw.x != xwend.x; ++xw.x)
                {
                    sum += sa.get(xw);
                }
            }

            // store average in destination
            da.set(sum / (winwidth * winheight), xd);
        }
    }
}
```

The algorithm calculates the average of all pixel values in a square window (who's size is given by `window_radius` in chess-board metric). If the window is partially outside the image, it is clipped accordingly. The result is written into the destination image. Note that `NumericTraits` are used to determine the type of the variable to hold the sum of the pixel values to prevent overflow (which would almost certainly occur if `SrcAccessor::value_type` were `unsigned char`).

This algorithm is easily generalized to arbitrary convolutions. Similarly to images, we pass convolution kernels by iterators and accessors. This allows us to represent kernels in many different ways — we can use dedicated kernel objects as well as images and even C-style arrays,

without requiring any change to the algorithm. In constrast to images, we must also pass the kernels' center. Therefore, the declaration of a general convolution function could look like this:

```
template <class SrcIterator, class SrcAccessor,
          class DestIterator, class DestAccessor,
          class KernelIterator, class KernelAccessor>
void convolveImage(SrcIterator sul, SrcIterator slr, SrcAccessor sa,
                   DestIterator dul, DestAccessor da,
                   KernelIterator kupperleft, KernelIterator kcenter,
                   KernelIterator klowerright, KernelAccessor ka);
```

Seeded region growing is a particularly good example for the versatility of the generic approach. In general, seeded region growing is defined by a set of seed regions, an image containing pixel information (like gray levels, colors, gradients etc.), and a statistics functor to determine into which region each pixel fits best. Thus, the corresponding function declaration would look as follows:

```
template <class SrcIterator, class SrcAccessor,
          class SeedIterator, class SeedAccessor,
          class DestIterator, class DestAccessor,
          class StatisticsFunctor>
void seededRegionGrowing(SrcIterator sul, SrcIterator slr, SrcAccessor sa,
                         SeedIterator seedul, SeedAccessor seeda,
                         DestIterator dul, DestAccessor da,
                         StatisticsFunctor statfunc);
```

Without any modification of this algorithm's source code, we can use it to implement many different region growing variants by simply passing different images and functors. Possibilities include:

- If the statistics functor returns always the same value, a morphological dilation of all regions will be performed until the image plane is completely covered. In particular, if the seed regions are single pixels, the Voronoi tesselation will be determined.
- If the source image contains gradient magnitudes, the seed image contains gradient minima, and the statistics functor returns the local gradient at every location, the watershed algorithm will be realized.
- If the statistics functor calculates and constantly updates region statistics for every region (such as mean gray values or colors), it can compare every pixel with its neighboring regions to determine where it fits best.

This illustrates a general experience we have made with genericity: Due to the versatility of generic functions, we do not need special implementations for every variation of a general algorithm. This means that a smaller number of generic functions is sufficient to implement functionality analogous to a traditional image analysis library. Also, generic functions are very robust under algorithm evolution: if better

growing criteria are found, we simply pass other functors and/or images, but the core implementation remains unchanged.

6.6 Performance

s:uk.performance

To assess the claims about the performance of generic algorithms we conducted a number of benchmark tests. We tested 4 different implementations of the averaging algorithm for a 2000×1000 float image with window size 7×7 on various systems. These are the implementation variants we tested:

1. a hand-optimized pointer-based version
2. a traditional object-oriented variant using the abstract image interface with a virtual access function as defined in `AbstractImage` in the introduction.
3. the averaging algorithm as shown in the previous section using the `AbstractImageIterator` which wraps the virtual function of `AbstractImage`.
4. the averaging algorithm as shown in the previous section using the standard implementation for image iterators operating on linear memory (see the CD-ROM).

The results of these tests are given in Table 6.2. The table shows that with the best compilers the proposed image iterator (variant 4) comes close to the performance of a hand-crafted algorithm (variant 1), whereas the versions using the abstract image interface (virtual function calls) are much slower (variants 2 and 3). Even in the worst case `ImageIterator1` takes less than double the optimal time. In the light of the flexibility gained this should be acceptable for many applications. Moreover, there is no principal reason that our iterator implementation must be slower than the hand-crafted version. Thus, with the permanent improvement of the compiler optimizers, we expect the performance of our iterator to increase further.

6.7 Image iterator adapters

s:uk.adapt

On the basis of the proposed iterator interface we can define iterators that modify the standard behavior in various ways. In many cases this allows to improve reusability by turning *algorithms* into *iterator adapters*. For example, algorithms that need one-dimensional projections of the image (i.e. a particular order in which all or some pixels are visited) can be generalized by encapsulating the particular projection into appropriate iterator adapters. By exchanging iterator adapters,

Table 6.2: *Performance measurements for different implementations of the averaging algorithm*

	Implementation Variant			
System	1	2	3	4
1	2.75 s (100%)	12.6 s (460%)	7.5 s (270%)	3.3 s (120%)
2	9.9 s (100%)	42.2 s (425%)	35.6 s (360%)	18.5 s (190%)
3	11.5 s (100%)	64.8 s (560%)	55.7 s (480%)	17.0 s (150%)
4	8.1 s (100%)	38.4 s (470%)	13.5 s (170%)	9.1 s (112%)
System 1	SGI O2, IRIX 6.3, SGI C++ 7.1			
System 2	SGI INDY, IRIX 5.3, SGI C++ 4.0 (outdated compiler)			
System 3	Sun SparcServer 1000, Solaris 5.5, Sun C++ 4.2			
System 4	PC Pentium 90, Windows NT 4.0, Microsoft VC++ 4.0			

t:uk.variants

different access sequences can be realized without changing the actual algorithm.

The simplest one-dimensional iterator adapter is, of course, the scan line iterator which scans the entire image in scan line order. It can often be implemented as a normal pointer to the raw image data which makes it very fast. Thus it is best suited for internal copy and initialization operations, when we need to process the entire image very fast.

6.7.1 Row and column iterators

s:uk.adapt.lines

Row and column iterators are a very useful kind of iterator adapters because they support the implementation of separable algorithms. They provide an interface compatible with the STL's random access iterators and restrict navigation to the specified row or column:

```
template <class ImageIterator>
struct RowIterator
{
    RowIterator(ImageIterator const & i)  // position of iterator i
    : iter(i)                             // determines row to operate on
    {}

    RowIterator & operator++() {
        iter.x++;     // go always in x direction
        return *this;
    }

    value_type & operator*() {
        return *iter; // delegate access to enclosed image iterator
    }
    // etc...

    ImageIterator iter;
};
```

Now, when we want to define a *separable algorithm*, just one implementation is sufficient for both directions (and in fact for any one-dimensional projection), because the actual direction is specified solely by the type of the iterator adapter passed to the algorithm. Consider the following one-dimensional algorithm for a recursive exponential filter:

```
template <class Src1DIterator, class SrcAccessor,
          class Dest1DIterator, class DestAccessor>
void recursiveSmoothSignal(
           Src1DIterator sbegin, Src1DIterator end, SrcAccessor sa,
           Dest1DIterator dbegin, DestAccessor da, float scale)
{
   ...
}
```

If we recall that this algorithm can operate in-place (i. e., source and destination signal can be identical, so that no temporary signal must be generated), we can implement the two-dimensional recursive filter by simply calling the 1-D version twice, first using row iterators, then using column iterators:

```
template <class SrcImageIterator, class SrcAccessor,
          class DestImageIterator, class DestAccessor>
void recursiveSmoothImage(
        SrcImageIterator sul, SrcImageIterator slr, SrcAccessor sa,
        DestImageterator dul, DestAccessor da, float scale)
{
    int width = slr.x - sul.x;  // calculate size
    int height = slr.y - sul.y; // of image

    DestImageIterator destul = dul;  // save iterator for later use

    for(int y=0, y<height, ++sul.y, ++dul.y, ++y)
    {
        RowIterator<SrcImageIterator> sr(sul);  // create iterator
        RowIterator<DestImageIterator> dr(dul); // adapters for row

        // convolve current row
        recursiveSmoothSignal(sr, sr+width, sa, dr, da, scale);
    }

    for(int x=0, dul=destul; x<width; ++dul.x, ++x)
    {
        ColumnIterator<DestImageIterator> dc(dul); // adapter for column

        // convolve current column (in-place)
        recursiveSmoothSignal(dc, dc+height, da, dc, da, scale);
    }
}
```

6.7.2 Iterator adapters for arbitrary shapes

s:uk.adapt.shapes

The idea of row and column iterators is easily generalized to arbitrarily shaped linear structures such as lines, rectangles, ellipses, and splines. For example, we can turn Bresenham's algorithm for drawing a straight line into an iterator adapter like this:

```
template <class ImageIterator>
struct BresenhamLineIterator
{
    BresenhamLineIterator(ImageIterator start, ImageIterator end)
    : iterator_(start), x_(0.0), y_(0.0)
    {
       int dx = end.x - start.x;  // calculate the distance
       int dy = end.y - start.y;  // from start to end
       int dd = max(abs(dx), abs(dy)) > 0 ? max(abs(dx), abs(dy) : 1;
       dx_ = (float)dx / abs(dd);  // init the appropriate
       dy_ = (float)dy / abs(dd);  // increments (Bresenham algorithm)
    }

    BresenhamLineIterator & operator++()
    {
        // calculte the next pixel along the line and advance
        // ImageIterator accordingly
    }
    // ...

    bool operator==(BresenhamLineIterator const & r) const {
        return iterator_ == r.iterator_;  // delegate comparison to
    }                                     // internal image iterators

    ImageIterator::PixelType & operator*() {
        return *iterator_;  // likewise delegate access
    }

    ImageIterator iterator_;
    float x_, y_;
};
```

A general drawing algorithm would now look like this:

```
template <class Iterator, class Accessor, class Color>
void drawShape(Iterator i, Iterator end, Accessor a, Color color)
{
    for(; i != end; ++i)    a.set(color, i);
}
```

By passing different iterators, this algorithm can draw any kind of shape. Similarly, we can implement an algorithm to output the intensity profile along an arbitrary contour, and we can even apply the linear convolution defined above to any curved line, always using the same set of iterator adapters to determine the curve.

There are many more possible iterator adapters which we can not describe here. To mention just a few: chain code iterators proceed according to a given chain code. *Contour iterators* follow the contour of a labeled region in a segmented image (see Fig. 6.2). Region iterators scan all pixels in a given region, which lets us calculate region statistics or fill the region with a new color. Interpolating iterators can be placed on non integer positions and return interpolated image data there.

6.8 Conclusions

s:uk.conclusions

The present chapter introduced a generic programming approach to the development of reusable algorithms in image processing and anal-

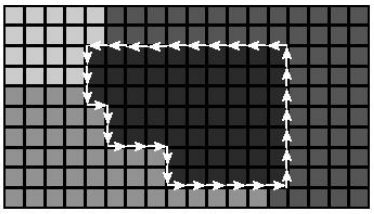

Figure 6.2: *A ContourIterator (white arrows) follows the contour of the darkest region*

f:uk.adapt.contour

ysis. It is based on ideas made popular by the C++ Standard Template Library and extends its abstract iterator design to the domain of 2-dimensional images. This approach overcomes a number of limitations of traditional ways to design reusable software:

- Programming is more efficient than with traditional methods since a single algorithm implementation can be adapted to many different situations.
- There is only a small performance penalty compared to an optimized version of the algorithm. Once the optimizers catch up, it is expected that the performance penalty can be eliminated altogether.
- A smooth transition to generic programming is possible: Iterators can be implemented for existing data structures. Switching to generic programming does not influence any existing code. Existing code can be transformed into a generic form as needed.
- Implementing generic algorithms is reminiscent of the traditional structured programming paradigm which many researches in computer vision are already familiar with.

Besides the results reported here, we have successfully applied the same techniques to build and manipulate graph based representations of image contents, such as feature adjacency graphs, feature correspondences across image ssequences, scene graphs, and geometric models. Our experience also showed that a relatively small number of iterator, accessor and adapter implementations suffices to support a wide variety of different environments, image data types, and applications.

It has proven very effective to combine generic methods with other software design approaches such as component based development (Chapter 4) and object-oriented design (Chapter 5). The basic idea here is to use objects and components to arrange functionality and to represent their relationships according to the needs of a particular application, while most of the real work is delegated to generic building blocks

residing below. In this way, strengths and weaknesses of the different methods are optimally balanced.

For example, in a digital photogrammetry system under development in our institute, we have wrapped generic algorithms and data structures into an object-oriented class hierarchy. In contrast to ealier designs, a reorganization of this class hierachy did not affect the algorithms' implementations — the compiler automatically generated the new versions needed by the revised objects, thus saving us from a lot of work. In another project, we similarly wrapped generic building blocks into PYTHON modules. PYTHON[7] is a public domain scripting language which can be extended by C and C++ libraries. Thus, with minimal additional effort we were able to build a simple (command driven) interactive image processing system on top of the generic VIGRA library.

Using the described approach, change and reuse of image processing and analysis algorithms has become much easier in our organization. Not only has programming productivity increased, but, perhaps more importantly, the psychological barrier to modify existing solutions has been lowered because it became so much easier and safer to adapt the existing code to new ideas. It would be highly desirable to officially standardize a common set of iterators and accessors as to prevent any compatibility problems between implementations that may otherwise arise.

6.9 References

uk01j94 [1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

uk02j96 [2] K. R. Inc. Khoros documentation (http://www.khoros.unm.edu/), 1996.

uk03j97 [3] D. Kühl and K. Weihe. Data access templates, c++-report july/august, 1997.

uk07j96 [4] S. Meyers. *More Effective C++*. Addison-Wesley, 1996.

uk06j96 [5] D. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

uk04j89 [6] D. Musser and A. Stepanov. Generic programming. In *1st Intl. Joint Conf. of ISSAC-88 and AAEC-6*. Springer LNCS 358, 1989.

uk05j94 [7] D. Musser and A. Stepanov. Algorithm-oriented generic libraries. *Software - Practice and Experience*, 24(7):623–642, 1994.

uk08j95 [8] N. Myers. A new and useful template technique: Traits, c++ report, june, 1995.

[7] See the PYTHON Web-Site at "http://www.python.org".