JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

INF
Damit Deine Zukunft in Zukunft Zukunft hat

A generic implementation of the data/view model

Diplomarbeit zur Erlangung des akademischen Grades „Diplom-Ingenieur“

im Diplomstudium Technische Mathematik

Angefertigt am Institut für Wissensbasierte Mathematische Systeme

Eingereicht von:

Roland Richter

Betreuung:

Univ.-Prof. Dr. Erich Peter Klement

Beurteilung:

Linz, Oktober 2004

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Roland Richter

PREFACE

When I first attended a C++ class back in 1994, the focus of the lecture was on the then still new idea of object orientation. It was not until some years later—and some other computer language classes passed, such as Fortran, Java, LISP, and Prolog—, that I realized that the C++ language had more to offer than just object orientation.

When I started to work at Fuzzy Logic Laboratorium Linz (FLLL), for me this meant the shift from theoretical examples to practical, every-day work with these languages and the related libraries and techniques. The more I got used to STL's container-iterator-algorithm style of programming, the more I strived to reformulate certain problems from our institute's main development focus – fuzzy computation and image processing – in terms of this new paradigm.

It was this desire which triggered my interest in, and inspired the work on an implementation of the data/view model. I decided to try to design a "view" library in the style and spirit of the Boost collection of C++ libraries. One detail, but a major one, was how to "group" or "zip" elements together in an appropriate way; in other words, how to form tuples. This work resulted not only in another library, but also in the further interest on topics such as generative programming. The theoretical background—everything I learned during the work on these libraries—forms the backbone of this thesis.

The two libraries—Boost.View and Boost.Tupple—are now available at the Boost sandbox, the playground for not-yet-reviewed potential Boost libraries. For more information on the Boost libraries in general, and on how to obtain the main libraries and the sandbox, please visit http://www.boost.org.

I would like to thank many people who, directly or indirectly, contributed to this thesis. Many thanks go to the Boost community; most notably to those who bothered to discuss design issues and implementation details, such as David Abrahams, Jaakko Järvi, Gary Powell, Vladimir Prus, and Martin Weiser.

The ideas of the Tupple library were developed mainly during a CEEPUS visit at Politechnika Krakowska. Thanks to Professor Piotr Kulczycki and Alek Mazgaj for a nice and productive stay.

Finally I'd like to thank Prof. Peter Klement for providing the environment and freedom to encourage such work; several colleagues at Software Competence Center Hagenberg, in particular Ulrich Bodenhofer and Markus Mittendorfer; and the whole staff of the FLLL.

Linz, October 2004
Roland Richter

CONTENTS

1. THE DATA/VIEW MODEL

> If I must apply a descriptive label, I use the phrase 'multiparadigm language' to describe C++.
>
> - Bjarne Stroustrup[1]

1.1 The evolution of C++

Over the years, the C++ language evolved from its predecessor C, which was limited to the imperative paradigm, to a multi-paradigm language which allows imperative, object oriented, generic, and functional style of programming.

We will consider one basic and important task, namely the element-wise application of a function to the elements of a collection, to demonstrate how the evolution of C++ enabled new forms of programming: from the simple **for**-loop to the container-iterator-algorithm style of the Standard Template Library to state-of-the-art "smart iterators".

Doing this, it will become obvious that there is still one further step to be done: namely to add "smart containers" in order to simplify the use of smart iterators. This so-called data/view model not only provides a means to replace explicit loops: it extends the generic programming paradigm of C++.

So here is our example which will guide us from the old imperative paradigm to generic programming, to the use of smart iterators, and beyond:

> Given a collection of elements, and a function which acts element-wise, apply the function to all elements of the collection.

1.1.1 The imperative paradigm

C was clearly an imperative language. With the transition from C to C++, object orientation was added to the scope of the language. Although from time to time there were discussions on C++ not being an object oriented language, it is nowadays widely accepted that C++ allows object oriented style of programming.

[1] Bjarne Stroustrup, *Why C++ is not just an Object-Oriented Programming Language*, Addendum to OOPSLA95 Proceedings, ACM OOPS Messenger, October 1995.

However, imperative style is probably still the most common one used when solving our little example. Hence, in C or C++ (or Java), the usual way is to write a loop which iterates over all the containers' elements, and would more or less look like this:

```
ElementType source[N], result[N];

for( int i = 0; i != N; ++i )
  result[i] = function( source[i] );
```

It's basic, simple, often used – so why looking for something different? Here are some reasons[2]:

- It relies on certain assumptions concerning the container type. In our case, both containers must provide so-called *random access* via `operator[]`. This is not true for associate containers, trees and the like.
- It is error prone. A loop as above relies on `i` being not changed within the loops' body. `N` has to be the correct size of the source container, and the size of the target container must be at least `N`. If one of these constraints is violated, undefined behaviour and program crashes are the likely consequence.
- It obscures what's happening. The first thing one sees when examining code like that is – well, that there is a loop. Only a closer look at the loop's body does reveal what actually is done; this might be difficult if the body is longer than a few lines.

1.1.2 Language elements

The above code fragment has a certain structure, and there are several different elements which make up this structure.

- Two *containers* – containing the source and the result elements.
  In typed languages such as C/C++, containers are usually intended to contain elements of *one* certain *type*. Therefore, containers can be characterized by their structure (random access, associative, tree-like, etc.) plus the type of their elements.
- A *function* – which maps elements from source to target.
  Its argument type must match the element type of the source container, whereas its result type must match the element type of the target container. ("To match" here does essentially mean that the later type is convertible into the former one).
- An *access mechanism* – how to get elements out of containers.
  In our case, this is done via an index and the (built-in) element access `operator[]`. It is assumed that we can use indices – which requires a certain type of container.

[2] For more criticism on external iteration, see, for instance, [Küh99, Chapter 10].

- A *"for all ... do" mechanism* – how to tie everything together.

It should be noted that for those who do not mind using plain pointer arithmetic, the statement above can be rewritten like this:

```
ElementType* r = result;
for( ElementType* s = source; s != source + N; ++s, ++r )
  *r = function( *s );
```

Sure, this does not make the loop more readable – just the opposite – but it is essentially the same and does not modify the underlying structure. The four language elements which we identified above are still present. Note that this way of writing things down inspired the iterator syntax which was later introduced within the STL.

1.2 Towards a new paradigm

Regarding the C++ programming language, the last decade was dominated by a long-lasting standardization process. It was finally brought to a successful end with the publication of the C++ Standard in September 1998 and the release of the first fully standard conforming C++ compiler *gcc-3.0* in June 2001. The *Standard Template Library* (STL), a C++ library which provides container classes, iterators (into these containers) and algorithms, is part of the standard. Its design and the wide-spread use of the STL introduced a new paradigm, which is nowadays usually called *generic programming*. This was only possible due to the introduction of a new language feature, namely the *template mechanism*[3]. The techniques which templates made possible were the focus of heavy research efforts during the last years, and it seems that their immense power is still not fully understood yet.

In the following, we will describe three important steps which influenced the way how to solve our example:

New container classes: Blank C arrays are stupid and error-prone. One of the most important features of STL was the introduction of various container classes plus their corresponding *iterators*. Iterators can be thought of as pointers into the container, pointing to one element at a time. In fact, the syntax of STL's iterators is as close as possible to plain C pointer arithmetic (and in simple cases, such as for std :: vector, an iterator is nothing else than an element pointer).

As a consequence, one of our language elements, namely the access mechanism, has changed. No longer we are limited to use indices; it is as well possible (and usual) to use iterators, which will result in code like this:

```
vector<ElementType> source(N), result(N); // or another STL container
```

[3] According to an interview given by Alex Stepanov which appeared in Dr. Dobb's Magazine in March 1995. Taken from http://www.sgi.com/tech/stl/drdobbs-interview.html

```
iterator res = result.begin();
for( iterator src = source.begin(); src != source.end(); ++src, ++res )
  *res = function( *src );
```

Functions as first class objects: Yes, in C it was possible to assign functions, to pass functions as other functions arguments, to write functions which returned functions – everything in the form of function pointers, which made the code rather hard to read [KR88].

However, these functions pointers required an exact function signature (for instance, a function of type `int f(int)` was not convertible to one of type `int f(double)`, although `int` was convertible to `double`), which resulted in the excessive use of `void*` arguments. Binding an argument of the function, or composing was difficult.

The template mechanism of C++ enabled a much more convenient and safe way to handle functions. In the sequel, the STL introduced the notion of *function object* or *functor*: a function object is simply anything that can be called as if it were a function, including ordinary functions, function pointers and objects of a class which defines `operator()`. An *adaptable function object* additionally provides `typedef`s to identify its argument and result types. This makes it relatively to easy to define meta-functions which operate on functions to give other functions, in STL called *function object adaptors*, for instance `bind` and `compose`.

What was left to be done was to link containers and their iterators to function objects. In STL, this is done via *algorithms*, and they became the third major component of STL. This step affected two language elements: instead of a function, we have a function object; and, as a consequence, the loop is now replaced by an STL algorithm:

```
transform( source.begin(), source.end(), result.begin(), function );
```

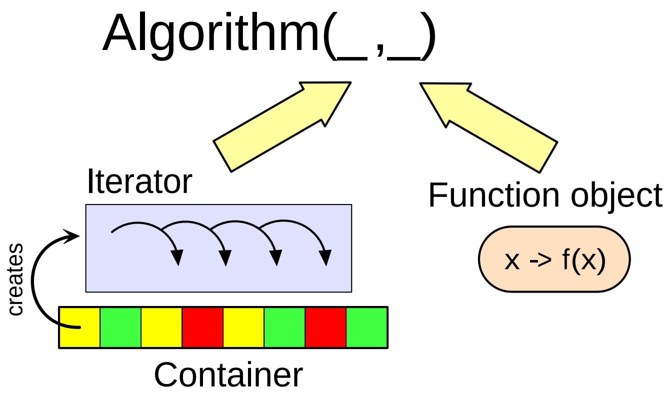

Figure 1.1. The layout of “classical” STL programming: containers, iterators, and functions are glued together via algorithms.

Smart iterators: Iterators are used if one needs access to sequentially organized data, typically data which is stored within a container (or array). However, sometimes we don't want to use the original data, but want to transform it (i.e. to apply a function), to select some parts of it, to reorder its elements, to combine it with other data, etc. before going on. The usual way to do this is to create another container, to iterate over the old container, transform/select/reorder/combine the iterated elements, and to store the resulting data in the new container (which, in turn, provides its own iterators).

This is somehow tedious, and we might do better. Following the widespread use of iterators, there were several attempts to add more functionality to iterators, i.e. to combine the iteration mechanism ("start at the first element – go step by step from the current element to the next one – return the current element – until the one-past-the-end element is reached") with additional functionality.

For instance, a "smart iterator" might read as "start at the first element – go step by step from the current element to the next one – *apply a function to the current element and return the result* – until the one-past-the-end element is reached" (a so-called *transform iterator*) or as "start at the first element – go step by step from the current element to the next one, *but ignore all elements which do not satisfy a given predicate* – return the current element – until the one-past-the-end element is reached" (a so-called *filtering iterator*) etc. Two approaches to provide smart iterators are [BB00] and the iterator adapters library of Boost, see http://www.boost.org.

When iterators get smart, STL algorithms are replaced by simpler ones. The functionality is transferred to the smart iterator, which has to be initialized with a "dumb" iterator and a function:

```
copy( transform_iterator( source.begin(), function ),
        transform_iterator( source.end(), function ),
        result.begin() );
```

1.3 State of the art: a multi-paradigm mix

So there are we now: We have better containers at hand. Their design both resulted in the replacement of arrays by container classes and pointers by iterators. Functions became first class members of our language. Iterators became "smart", and we find ourselves in a multi-paradigm environment. We started with an imperative style and introduced more and more of the generic paradigm, which, paradoxically, allowed us to write programs which look like functional style.

Of course, we know that the paradigm one uses does strongly influence the way how to solve a specific problem, how efficient the implementation is (and how efficient it can be at best), how neatly it fits into the language framework, how reusable and, last but not least, how readable it is.

What is there left to be done?

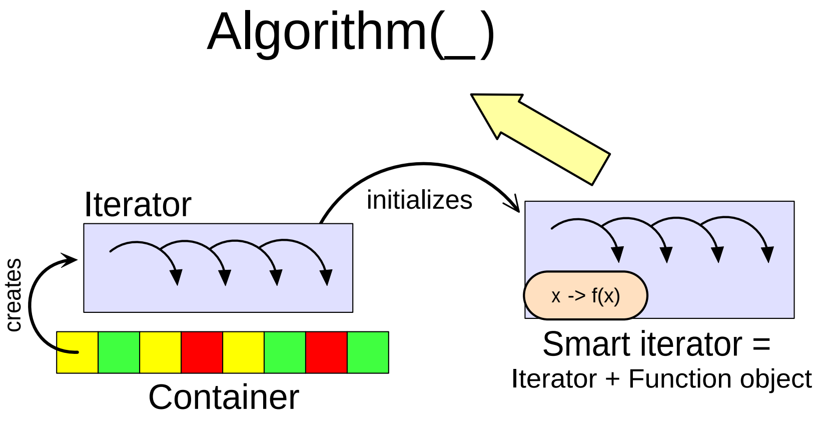

FIGURE 1.2. The layout if smart iterators enter the scene: the function object is now part of the smart iterator.

1.4 One further step

Although it is not a completely perfect analogy, the next step might be described as making containers "smart", too [4].

In terms of iterators, getting smarter meant that, instead of simply iterating over an already existing collection of elements and passing one after the other, smart iterators are able to create elements on the fly, to skip certain elements, to change the order of elements etc. A smart iterator wraps a "dumb" iterator and adds some functionality. Smart iterators do not modify their contents; however, from the outside it appears *as if* they modified it.

Along the same principles we might design "smart" containers: instead of simply containing elements, they should create them just-in-time, hide some of them, modify the ordering in which the elements appear, and so on. A smart container should wrap a "dumb" container and adds some functionality. Again, this does *not* mean that a smart container should modify its underlying container. These containers act *as if* they modified their contents, without actually doing it.

In short, we can identify the two sides from which we can see such a smart container:

- inside, there is the underlying, unmodified *data*
- from the outside, we only can see a modified *view* of the data

Therefore the name – the data/view model. We might summarize the facts on views up to now and try to give some sort of a definition of views. It might be like this: "A view does wrap a container and presents the container's data to its clients. The presented data will be some modified version of the original one; however, the underlying data is not modified in any way; it just appears from the outside as if it was modified. Views provide an interface which is as close as possible to that of the wrapped container class."

[4] A completely different solution is provided in [Küh99] called the transfold pattern.

1.4.1 Historical notes

The term "view" occurs in computer science at different places, most often in the sense of "encapsulate the knowledge how to present some underlying data", for instance with the Smalltalk Model/View/Controller triad.

The first to introduce views in our sense was Jon Seymour [Sey95] in 1995. His work was then inspired by the analogy to relational views of relational database theory.

The yet most complete implementation of views is the View Template Library by Powell and Weiser [PW99][WP00].

1.4.2 Various interpretations of views

Views as container proxies/decorators: Lets try to classify views in term of design patterns as introduced by [GHJV95]. By construction, views are meant as a stand-in replacement for STL container classes. Especially, they should provide exactly the same interface as containers. However, they do add some functionality: elements of the underlying container are presented in some modified way.

Speaking in terms of design patterns, views as placeholders for containers might be seen as instances of the Proxy pattern. Views as classes which attach additional functionality to container classes might be interpreted as the static form of the Decorator pattern. In the words of [GHJV95], "a decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object". Well, a view does both (in most cases).

More precisely, views can be seen as *virtual* proxies since they create objects on demand; as *protection* proxies because they control access to the underlying containers elements and usually prohibit write access.

Originally, decorators are described to attach functionality *dynamically*, i.e. at run-time. Views are not a dynamic, but a static concept: they change the behaviour of the underlying class at compile-time. Still, the analogy seems justified, especially since the implementation of decorators and the implementation of views show some remarkable parallels.

One must not get confused by the frequent use of the term "adaptor"[5] in the STL world, as opposed to "Adapter" in the sense of design patterns. The term "adaptor" was first introduced with the concept of function object adaptors in the STL, and later used for various other concepts, for instance in the term "iterator adaptor". Its meaning might be described as "wrap an existing object, do add some functionality, and provide the same interface as the underlying object's class". Contrary, an "Adapter", as described in [GHJV95], "converts the interface of a class into another interface clients expect". "Adaptors" usually *do not* change interfaces. Hence, most often, the term "adaptor" might be translated as "static decorator".

Views as smart iterator factories: Since views do mimic STL container classes as close as possible, they certainly have to provide methods `begin()`

[5] Interestingly, always written with an "o": adaptor.

and `end()`, i.e. methods to return smart iterators to the first and one-past-the-end element, respectively.

The concrete type of the smart iterator depends, of course, on the type of the view. Different views of the same container can give different smart iterators. Insofar, as much as STL containers can be seen as factories of iterators, views can be seen as factories of smart iterators (compare [WP00]).

Views in relational database theory: Another, quite different way to think about views is to consider them as an analog of a relational view in relational database theory, as did Jon Seymour [Sey95] in his original work.

1.5 Characteristics of views

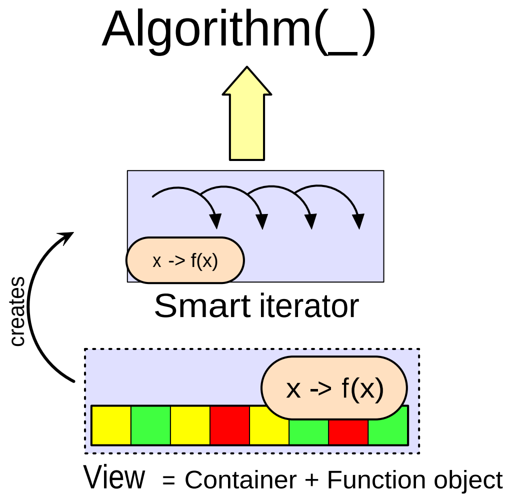

Figure 1.3. The data/view model: since a view contains information on both the underlying container and the function object, it can generate the corresponding smart iterator.

In some sense, with “classical” STL-like containers, iterators and containers are bound closely together, whereas the function is only coupled via STL algorithms. Smart iterators change the picture in that they contain the function object; still, they have to be initialized using an ordinary iterator. In contrast, with data/view-like containers, functions are integrated into containers, which no longer produce dumb, but smart iterators. These “mechanics” of these three techniques are depicted in figures 1.1, 1.2, and 1.3, respectively.

1.5.1 When to use views (and when not)

In short, views might be used

- to replace explicit loops: Our original motivation was to reconsider the use of a loop. However, the STL already did that: it replaced explicit loops by more specific algorithms. Therefore, it is justified to go one step further and identify a second use of views:

- to replace STL algorithms: Indeed, tasks like transformation of data, filtering, reordering etc. all find their corresponding algorithm in the STL; in the sequel, some STL algorithm do_something will be replaced by the corresponding do_something_view. It is, however, not possible to replace all algorithms, and the reason will become clear in a moment.

As a rule of thumb,

- views are appropriate whenever the function is local, i.e. "element-wise" or restricted

- views are *not* appropriate whenever the function requires global information A simple example would be to find out the maximum element of a container: we can't calculate the maximum without considering every element of the container, that is, maximum is no local operation. The same is true for sorting, etc.

1.5.2 The interface of a view

Saying that a view's interface should be as close as possible to the interface of a STL container does enclose some vagueness. Enumerating what the interface of a view must in fact contain results in the following list:

1. Constructors which take a container as argument
2. Destructor
3. Copy constructor, assignment, swap()
4. begin(), end()
5. rbegin(), rend()
6. size(), max_size(), empty()
7. **operator**[]
8. Various typedefs
9. **operator**== (this implicitly defines **operator**=! as well)
10. **operator**< (**operator**>, **operator**<=, and **operator**>= are all implemented in terms of **operator**<):
11. Usually *not*: insert(), erase(), clear() and the like

1.5.3 Container ownership

As we already mentioned, a view does not exist on its own. Instead, they are only adaptors of their underlying containers. The question arises how this relationship should be designed, i.e. how a views does “own” its underlying container. There are three possibilities to design this relationship:

a) As a copy of its own: The safest way to handle container ownership is to keep a copy of the container within each view. This would be appropriate, for instance, if we plan to pass views around, or if we want to keep a view even if the underlying container is going to be destroyed. However, as a consequence, whenever the view is constructed, copy constructed, or assigned, we have to do a deep copy, which might be quite time-consuming.

b) As a reference (or pointer): The cheapest way (in terms of both time efficiency and memory consumption) for a view to own a container is to hold a reference or pointer to it. In that case, one has to make sure that the view is invalidated whenever its container is destroyed to prevent dangling references (or pointers).

c) As a reference-counted copy: This is a compromise between safety and efficiency: copying is done only once, namely if the view is constructed with the container as an argument. In the case of copy construction and assignment, however, the reference count is simply increased, which a relatively cheap operation compared to copying the whole container.

1.5.4 Eager versus lazy evaluation

A view (and, also a smart iterator) does contain (or reference) its underlying container; it does not, however, contain the transformed elements it gives back when the element access method (**operator**[], or dereference **operator***, respectively) is called. Instead, the transformed elements are generated on the fly, i.e. only when needed. This behaviour is called *lazy evaluation* (as opposed to *eager evaluation*). It increases efficiency if only part of the data is used.

1.6 The functional paradigm

Only relatively recently, the C++ community undertook efforts to enable the use of the functional paradigm within C++. Already the STL introduced the notion of *function objects*, and hence made functions first class objects of the language. This lead to libraries implementing the functional paradigm (and even lambda calculus) to the full extent. The power of these features was explored by several libraries, for instance the FC++ library [SM01], and the Lambda library[SS00].

A program written in a pure functional language consists of just one thing – functions. Typically, functional languages provide a set of meta-functions which take functions as arguments and apply it in a certain way.

For instance, in Mathematica(TM), the function Map does apply a function to each element of a list; hence, our example might be solved like this:

```
result = Map[ function , source ];
```

Again, we can identify the major participants and compare them to those of the imperative paradigm:

- Two *containers* - typically built-in in functional languages
- A *function* - passed as an argument to a higher-level function
- A *meta function* - typically implemented via recursion

2. A CATALOGUE OF VIEWS

> There are many paths to the top of the mountain, but the view is always the same.
>
> - Ancient Chinese Proverb

2.1 Design issues and questions

The first chapter was devoted to the "philosophical" aspects of the data/view model, its place in the historical context, and how it fits into the generic programming paradigm.

In this chapter, we list a "catalogue" of individual view classes, and focus our attention onto the issues of the functionality and structure these individual view classes have, when and how to apply them, and how they interact.

Before proceeding, it is reasonable to consider some general issues which are of importance when dealing with views in general.

1. One point we have to ask is: does the view concern the *contents* or the *structure* of the underlying container? With "contents" we mean the values of the containers' elements; with "structure" its size and the arrangement of the elements. This difference will especially get clear in the discussion of permutation_view.

2. Most views take one container, some more than one. There is yet another type of views which, as an exception to the rule, do *not* reference to an underlying container at all.

3. Some views do allow read and write access to their elements (better: to the elements of the underlying container), others do prohibit write access and allow read access only. Of course it would be nice to design views such that element access is read and write, but in some cases there is simply no sensible way to do this. Further on, allowing to write elements also raises problems if more than one view does reference to the underlying collection.

   In general, views will be read-only unless there is a "straight-forward" way to allow writing.

4. Views are always closely connected to their (smart or adapted) iterators. As we will see later when thinking about how to implement views, the

most practical way is first to select the corresponding iterator, and only then to implement the view itself.

5. Some views are suitable to replace one (sometimes even more than one) STL algorithm; however, there seems to be no systematic relation between views an algorithms. "Views" and "algorithms" are two related, but yet different ways of how to solve certain problems.

6. One issue we will give focus when it comes to the topic of implementing views will be that of runtime efficiency. The C++ standard imposes runtime efficiency characteristics to STL container operations; sometimes views have a different characteristics.

The overview and terminology is in part extracted from the View Template Library (compare [PW99], [WP00]) and the work on iterator adaptors (see, for instance, [BB00] and [AS01]). Other parts were added. In some sense, this catalogue constitutes a "wish-list" for what one might expect as part of a view library.

2.2 Transform view

2.2.1 Overview

Given an underlying container and a transform function, a *transform view* presents a view of the container where each element is the result of applying the transformation function to the corresponding underlying element.

2.2.2 Description

Transform view might be regarded as one of the basic and most "typical" applications of the view concept. Its functionality (together with filtering, see SECTION 2.4) was the core of Jon Seymour's original work [Sey95], and transform_iterator is one of the basic usages of Boosts' iterator adaptor library.

A transform view does affect the contents of a container, not the structure. That is, it does operate on the values of elements, but their relative position within the collection, as well as the collections' size, is unchanged. Insofar, it is a little bit untypical, because most views which we will learn in the sequence do in some sense restructure the elements of the collection, but do not touch their original value.

Further, a transform view does *not* support write access to its elements. To make it writable, it would be necessary to calculate the inverse of the transform function, which, in turn, would force the function being bijective. Deciding that this is too much effort for too little gain, a transform view will always be read-only in our treatment.

2.2.3 Application

- Use it as a replacement of the transform algorithm of STL

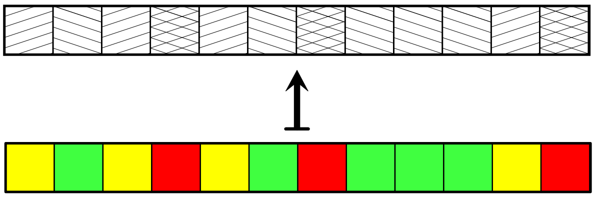

Figure 2.1. Transform view: the transform function is applied to each element.

- Transform views might be nested in the same way as functions might be
- The function of the transform view may either be fixed at compile time or be chosen at run-time
- Transform views might also be used to sub-sample a continuous function at a (possibly non-uniform) discrete grid

2.3 Permutation view

2.3.1 Overview

A *permutation view* applies a "re-indexing scheme" to the elements of the underlying container. A re-indexing scheme is, in its simplest form, just another container representing the new arrangement of the elements, containing the indices pointing into the underlying container. A more advanced usage is to generate the scheme "on-the-fly" via the use of another view.

There are very weak requirements which this scheme has to fulfil. Especially, the scheme is *not* necessarily injective, or surjective; it might be of different size than the underlying container; it might even have an index type different of that of the underlying collection. This flexibility makes it possible to solve a wide variety of tasks using a permutation view.

2.3.2 Description

In short, we might say that a permutation view takes the *contents* (i.e. the values represented by the elements) of one container and the *structure* (i.e. the position of the elements, expressed through their indices) of another container and merges both into a new container. The permutation view presents elements of the first container at positions determined by the second one.

In Figure 2.2, for instance, the first element of the view (i.e. that one with index 0) is element number eight of the original container (i.e. that one with index 7). The element itself comes from the first container, its position within the view is determined by the second container.

Note that it is valid to assign one and the same elements of the container to several positions of the view; for instance, former element number 3 now appears

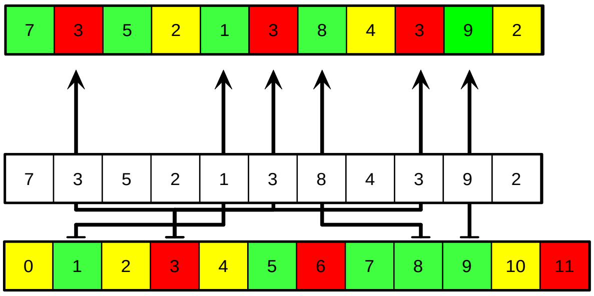

Figure 2.2. Permutation view: all elements are re-arranged according to a re-index scheme.

at positions 1, 5, and 8. On the other hand, some elements of the original collection are missing in the view, for instance, elements 10 and 11. Finally, the size of the view is not equal to the size of the collection; it might be smaller (as in our case), bigger, or even empty. From that it gets clear that the size of the view is that of the re-indexing scheme; and the view is empty if and only if the scheme is.

However, creating the re-indexing scheme by hand is somewhat clumsy, and might cost unnecessary time and space. In a way it is not "as lazy as it could be". The (obvious) way to overcome it is to provide the re-indexing scheme in form of another view, for instance as a function_view.

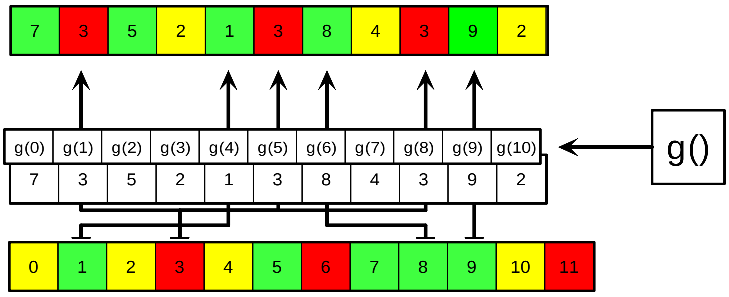

Figure 2.3. Permutation view: the re-index scheme might be provided via a re-indexing function.

A permutation view allows both read and write access; this is *not* in contradiction with the fact that the permutation might be non-injective.

2.3.3 Application

A permutation view might be used for one of the following tasks:

- to change the order of the elements of a collection; thus, it might serve as a replacement of STL's reverse algorithm.

- to select certain elements out of the container, or to select a range of elements from the container; note however that filter_view or window_view might be more appropriate.
- to change the type of indices of a container, as shown in FIGURE 2.4.

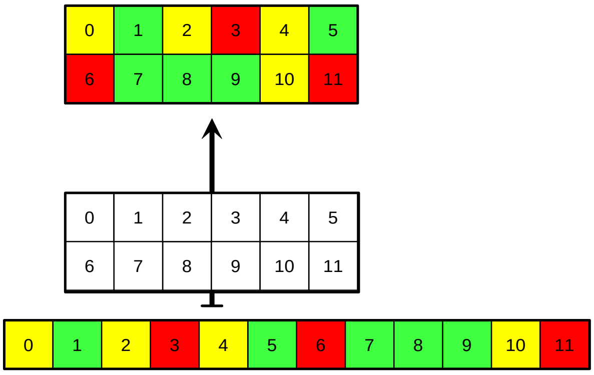

FIGURE 2.4. Permutation view: the re-indexing scheme might as well change the type of the index.

2.4 Filter view

2.4.1 Overview

A *filter view* is a view which is capable to select certain elements out of the given container. Given a predicate operating on the elements of the underlying container, i.e. a unary function which has the element type as argument type and returns a (type convertible to) **bool**, a filter view presents only those elements of the collection for which the predicate is **true**.

2.4.2 Description

In a strict sense, a filter view is just a specialization of permutation view, that is, the same functionality could also be achieved using a permutation view of the collection together with the set of indices of elements fulfilling the predicate.

Therefore, everything that we mentioned in the previous section is also true for a filter view:

- A filter view does affect the structure of a container, not the contents.
- A filter view allows both read and write access to those elements which fulfil the predicate. Be aware, however, of the strange situation when to an element of the filter view another value is assigned which does *not* fulfil the predicate.

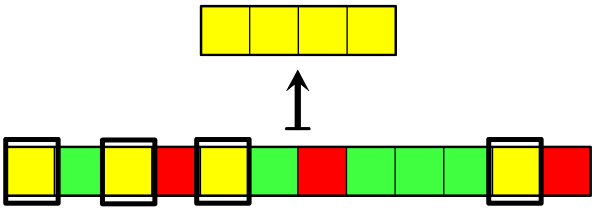

FIGURE 2.5. Filter view: only elements of a certain kind are shown.

Note that functions empty() and size () of filter view require to step through the whole underlying collection anyway, so both are linear in the container's size.

2.4.3 Application

- Use a filter view if you want to select certain elements of a container, depending on a predicate
- The use of a filter view might serve to replace several " ... _if " algorithms such as find_if , count_if , and replace_if .

2.5 Window view

2.5.1 Overview

A *window view* is a rather simple class intended to select a window, i.e. a range of the form [b,e) (where b and e are iterators) out of the underlying container, with the possibility to rotate the window through the container.

2.5.2 Description

Note that it is not necessary for the range to be valid range in the sense that is had to fulfil the condition that b is "less than" e, or e is *reachable* from b. If necessary, the selected range is "wrapped around", that means that iteration restarts at the beginning of the container if it went past the end.

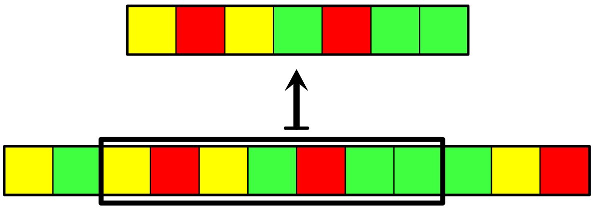

FIGURE 2.6. Window view: select a range out of a container.

For random access containers, it is also possible to specify a range in the form of [i,j), where i and j are the indices of the first and one-past-the-end elements, respectively.

Some additional methods allow to rotate the window through the container. If you don't need that functionality at all, consider using the simpler range_view (see SECTION 2.7) instead.

This view might be seen as another specialization of permutation view, therefore it does affect the structure of a container, not the contents, and it is both read- and writable.

2.5.3 Application

Use window view

- to select a sub-range out of a container
- to implement algorithms which operate on a certain *neighbourhood* of the current element – compare neighbour_view.
- to enable iteration past-the-end of the collection

2.6 Function view

2.6.1 Overview

A *function view* presents the view of a container where the elements of the container are generated by a function; iteration runs over a user-defined range.

2.6.2 Description

Up to now, we have several times stressed that a view is a proxy to a collection of elements (even if this is another view); that a view owns, or wraps, its underlying container and presents its elements in some modified way.

As always, there are exceptions to the rule: there are views which do *not* wrap any existing container; instead, they create a container-like structure "on the fly".

Function view is such a structure: instead of an underlying container, you have to provide a generating function and a range to enumerate the arguments of the generating function.

2.6.3 Application

A function view is appropriate

- to obtain a view of a container where elements are generated by a function
- to use instead of the generate algorithm of STL

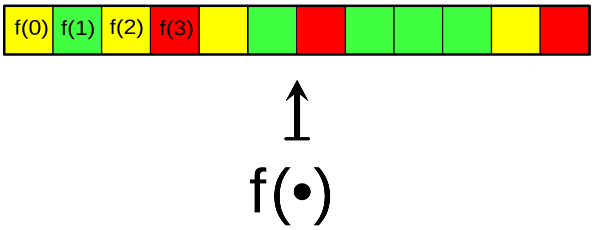

FIGURE 2.7. Function view: present a container-like view by means of a *generating function*.

2.7 Range view

2.7.1 Overview

Taking two iterators, a *range view* presents a container-like view of all elements between these two iterators.

2.7.2 Description

The other view class which does not require an underlying container is range view: two iterators at the begin and end are taken to mimic a collection which contains all the elements between first (included) and last (excluded) element.

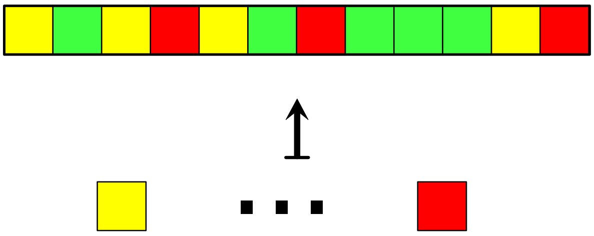

FIGURE 2.8. Range view: present a view of elements between the first and one-past-the-end iterators.

2.7.3 Application

- to avoid the need to explicitly construct and fill a container
- to create a view using some adapted iterator, for instance a `counting_iterator` .
- to select a range of an existing container (instead of a window view)

2.8 Chain view

2.8.1 Overview

Given several containers with identical iterator types, a *chain view* concatenates them all together. From the outside, the result looks like *one* single container.

2.8.2 Description

Many container implementations provide methods to append (or link, or concatenate) one container to the other. If one treats strings as character containers – and the std :: string implementation of STL does so –, string concatenation can also be considered as a chaining operation.

Assume, then, that you want to append a second container to the first one; then append the third to the result of joining containers one and two; then comes the forth, and so on. Of course, some of the containers might be empty. Others might be quite large. Do you know whether your append() function does copy the whole content of one container in order to append it to the other, and therefore wastes much time?

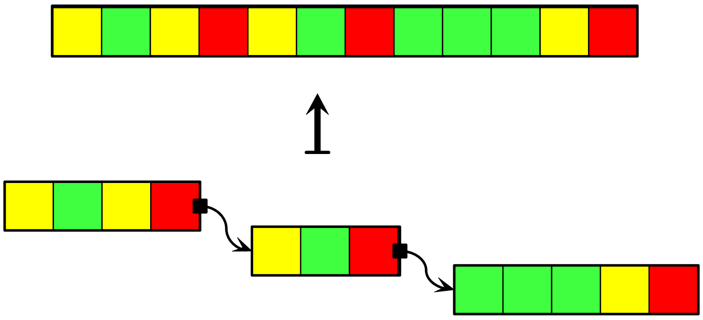

FIGURE 2.9. Chain view: Link containers together.

A faster alternative might be a chain view: instead of actually appending containers and creating a new one, it just lets it appear as if they were appended[1].

Of course, there is the modest price of memory that the view needs to do all the book-keeping. A chain view, however, is not just intended to save the time of appending operations. With the help of a window view, also insertion of one container into another might be simulated. Additionally, a chain view is able to reduce iteration complexity.

Assume that we have a large container, and that we want to iterate over certain parts of this container only. We already learned one way to do that, namely by using a permutation view. As it happens, our regions of interest are given in the form [begin,end), and we want to select the interesting ranges with a window view, as shown in FIGURE 2.10.

[1] You're probably getting used to this motif by now.

One solution is to iterate over the interesting parts using two nested loops, like this:

```
for( every region of interest )
  for( every element in the region )
  {
    do_something();
  }
```

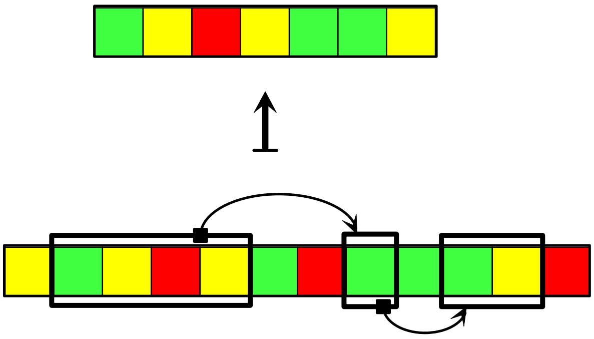

FIGURE 2.10. Select some ranges with a window view, then link them together with a chain view.

There is yet another possibility, however: if one links all the selected regions together using a chain view, these two loops are reduced to one; that is, we can write

```
// build a chain view of all regions of interest
for( every element in the view )
{
  do_something();
}
```

In other words, we have reduced two-dimensional iteration to one-dimensional iteration. Whereas it might not be obvious right now why this is a benefit, we will come back to this topic later, then considering iteration in image processing.

2.8.3 Application

- to link several containers together “on the fly”
- as a fast alternative to concatenation or insertion operations
- to reduce two-dimensional to one-dimensional iteration

2.9 Zip view

2.9.1 Overview

A *zip view* takes a fixed number of collections, say, n, and ties them together in the sense that it presents their elements as n-tuples.

2.9.2 Description

A rather common situation is to have two separate containers together with a binary function (or a binary predicate) which should be applied to the *pairs* of elements of container one and two. The trick is to provide a *unary* function taking *one pair* which does just the same as the *binary* function taking *two single elements.*

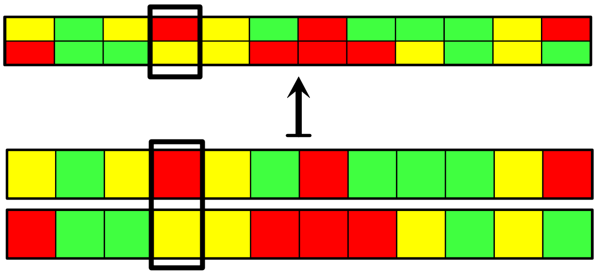

FIGURE 2.11. Zip view: present several containers as one container of tuples.

In other words, before applying the function, we want to "zip" or "glue" or "tie" together two collections. It should appear as if we had *one* collection containing *pairs* instead of *two* collections containing single elements. That is exactly what the zip view should do (at least temporarily).

The same game can be played, of course, with three containers and a ternary function, giving triples of elements, or, in general, with n containers, an "n-ary" function, and "n-tuples". Whereas the details of the implementation of tuples will be presented in one of the next reports, as a guiding principle we should keep in mind that tuples should be a generalization of STL's `pair` structure.

As simple and elementary this operation might seem, it is rises a bunch of questions. When containers are zipped together, so are their elements and their iterators. We have to take care that each operation performed on the zipped container is a proper extension of basic container operations. For instance, accessing the second element of a zip view of three containers means to access the second element of the first, of the second, and of the third container, respectively. Incrementing an iterator means incrementing iterators pointing into the first, into the second, and into the third collection, respectively – and so on: each operation on the zip view is an "element-wise" application of the underlying basic operation.

Another question concerns the situation if, for instance, the first container is empty and the second is not. Should we regard the resulting container as being empty?

2.9.3 Application

- to apply a binary, ternary, or n-ary function to a number of collections
- to group elements of different collections into tuples

2.10 Neighbour view

2.10.1 Overview

A *neighbour view* ties together several elements (the "neighbourhood") out of one container into a tuple.

2.10.2 Description

Assume our task is to compute the moving average of the given container; alas, you've got the transform_view at hand, but averaging is certainly a function depending on more than just the current element.

Moving averages, and other "local" algorithms, depend on what we call a *neighbourhood* around the current element; that is, its size (the number of elements) and the position of each element relative to the current one are *fixed.*

Neighbour view does group the elements of a neighbourhood into a tuple. Again, a fixed number of elements is collected together into a tuple; but this times, elements are all out of the *same* container.

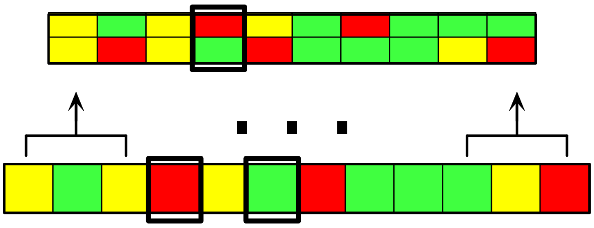

FIGURE 2.12. Neighbour view: tie together elements of a neighbourhood into a tuple.

2.10.3 Application

- to implement algorithms which rely on a neighbourhood around the current element

3. IMPLEMENTATION ISSUES

> If debugging is the process of taking bugs out, then programming must surely be the process of putting them in.
>
> - John Topley[1]

3.1 A view's interface

Remember that views are intended to be a drop-in replacement for STL container classes. Consequently, a view's interface should be as close as possible to the interface of such a STL container. The question is only – which sort of container? There are *Random Access Containers*, *Back Insertion Sequences*, and *Unique Sorted Associative Containers*, among others.

The short answer is that we will try to provide the interface of a Random Access Container – with a few exceptions, of course –, that we might provide some functionality of Back Insertion Sequences, and that we will deal with some workarounds imposed by STL's map. Of course, there are no rules without exceptions, and for some view types this simple answer is not applicable at all[2].

Following closely [Jos99, section 6.10], lets see which parts constitute a containers' interface. In general, we will skip any functionality which deals with modifying the container (but there will be methods to modify the containers' elements).

1. Type Definitions: a number of type definitions are sprinkled throughout the interface code, and it pays off to provide them in a systematic way. In fact, these **typedef**s deserve a section on their own – see SECTION 3.2.
2. Create, Copy, and Destroy Operations: in other words, constructors, copy constructor, and destructors.
3. Non modifying Operations:
   - Size Operations: size () and empty(), but not max_size().

[1] Quote taken from the thread "Where do bugs come from?" of the Joel on Software discussion forum (http://discuss.fogcreek.com/joelonsoftware/).

[2] These considerations also led me to the believe that there is no basic view class from which all other view classes can be derived. Views are just too different; defining a base class might be possible, but I believe this will not have any advantages at all.

- Capacity Operations: None. Views should not deal with the storage of the container, thus do not expect capacity() and reserve().
- Comparison operations: **operator**==, which implicitly defines **operator**=! as well. **operator**< to compare lexicographically; the operations >, <=, and >= are all implemented in terms of <.

4. Assignments: **operator**= and swap().

5. Direct Element Access: **operator**[index] and its more pedantic version, function at(index). The exact type of index will be subject to discussion in Section 3.2. Since front() and back() are useful, yet easy to implement, they will most often be part of a view's interface as well.

6. Operations to Generate Iterators: such as begin() and end(). In the case of views, the name "operations to *generate* iterators" is definitely justified. Views often will actually have to compute begin() and end(), whereas STL containers do usually just pass a stored pointer. I will *not* care about rbegin() and rend().

7. Inserting and Removing Elements: none at all. A view is a adapter to an existing container, and it won't modify its underlying container, so it should allow no modifying functions. No view will provide things such as insert(), erase(), clear(), and the like.

As a rule of thumb, a view is implemented in terms of its related smart iterator, if possible. A transform_view wraps a transform iterator and adds a container-like interface; a filter_view transfers all necessary computations to its filter iterator; and so on. Smart iterators also play a key role when defining all the necessary types for a view, as shown in the next section.

3.2 Type computations

In order to implement a view, we have to provide – or compute – several types which are related to the view itself, to its underlying container, or to any another component (such as a function or predicate) which the view is composed of.

As always, we stick to our paradigm: we want a view to be as similar to a STL container as possible. Each and any STL container already defines nine related types; as a consequence, each and any view has to provide them as well:

1. value_type: The type of the elements stored in the view.

2,3. iterator and const_iterator : The types which are used to iterate through the view's elements.

4,5. reference and const_reference: Types which behave as a reference to the view's elements.

6,7. pointer and const_pointer[3]: Types which behave as a pointer to the view's elements.

8. difference_type : A signed integral type; it is the type of the distance between two iterators.

9. size_type : A unsigned integral type; represents the number of elements stored in the view.

All these types are required to implement an STL container and its methods. For instance, a random access container provides access to its elements via the **operator**[] method, which looks as follows:

```
const_reference operator[]( convertible to size_type ) const;
```

With those nine types, we are *almost* done. There are just two exceptions. First, all associative containers associate their elements with keys, and hence have to define another type:

- key_type: The type of keys associated with the value type.

Pair associative containers introduce yet one further type. This kind of container stores key-data pairs; thus, not only the key's type, but also the data's type has to be specified:

- data_type: The type of the data associated with the value type.

In this case, value_type is *forced* to be pair<**const** key_type, data_type>. This already introduces a nasty inconvenience. STL's implementation of a sorted, unique pair associative container, class map< Key, Data >, also provides an **operator**[] – but it does it its own way:

```
data_type& operator[]( const key_type& k )
```

How can we provide *one* definition of **operator**[] without knowing in advance which of the two versions the underlying container will provide? We can't do so in a completely straightforward manner. My proposal here is to introduce two further types for *any* view:

10. index_type: the argument type of **operator**[]

11. data_type: the result type of **operator**[]

Eleven type definitions all in all! Where do all these types come from? The short answer is: compute them whenever possible.

[3] There has been a lively discussion on comp.lang.c++.moderated on whether const_pointer is defined or not (March 2003). In short, the standard does *not* list const_pointer as part of the container requirements. However, it is implicitly introduced through their allocator template argument.

First and fore-most, each view defines its own smart iterator type, which in turn is obtained by applying some iterator adaptor. Using Boost's iterator_adaptor library, we are in luck: many types are already defined in there, and all a view has to do is to extract them out.

Other types of the view are usually dependent on its underlying container. Sometimes, they are just identical to the containers' types; sometimes not. In general,

- size_type, index_type, and data_type are derived from the views' container type, whereas
- all other types are extracted from the views' iterator type.

We want to extract the types without fiddling around with the view's code too much. This can be achieved with a technique called *traits*. As subsumed in [Mye95], a traits class "provides a convenient way to associate related types, values, and functions with a template parameter type" – and that is exactly what we need.

So, all we have to do is to add two (private) traits type definitions – one for iterator types, and one for container types – to our my_view implementation, like this:

```
typedef traits<
          my_iterator_generator<several types>::type,
          my_const_iterator_generator<several types>::type,
        > iter_traits;
```

These are then used to actually provide all types "in public":

```
typedef typename iter_traits::value_type       value_type;
typedef typename iter_traits::iterator         iterator;
//...
typedef typename cont_traits::size_type        size_type;
//...
```

So, the "surface" of a general view class is fixed now; lets have a look at the innards.

3.3 Ownership

3.3.1 Motivation

Most views contain one (or more) underlying STL containers. However, what does "containing" mean in this context? A naive interpretation might be that "contains a STL container" is the same as "has a STL container as a (private) member". This leads to a first implementation skeleton of a view, which looks like Listing 3.1.

```
template<class ContainerT> class my_view
{
  // The constructor.
  my_view( const ContainerT& theData ): data( theData ) {}

  // The copy constructor.
  my_view( const my_view& other ): data( other.data ) {}

  //...
private:
  ContainerT data; // View contains its data as private member.
};
```

Listing 3.1. Code skeleton of naive view implementation.

This is simple and does what it is expected to do; at the price that construction, copy construction, and assignment all require a deep copy of the container, which might be quite time-consuming. To illustrate this point, let us consider the case of "viewing a view", as shown in Listing 3.2.

```
any_container u; // any container                                      1
                                                                       2
my_view<any_container>                a( u ); // view of container         3
my_view< my_view<any_container> > b( a ); // view of view of container     4
```

Listing 3.2. Construct a view and a view of a view.

It is important to note that `a` and `b` are *not* of the same type; `a` is a view of an ordinary STL container, whereas `b` is a view of a view. Hence, in both cases the ordinary constructor is called.

More precisely: if view `a` is constructed in line 3, its constructor is called:

```
  my_view( const ContainerT& theData ): data( theData ) {}
```

This does call the copy constructor of the STL container, which does usually a deep copy of the containers' contents.

The template argument ContainerT of my_view is instantiated to my_view<any_container> in line 4. Hence, b's member data is of exactly that type. When we call b(a) in line 4, the ordinary constructor of type my_view< my_view<any_container> > is called; this, in turn, calls the copy constructor of my_view<any_container> – and that does again copy the container. Finally, we end up with the situation of Figure 3.1: each view contains its own copy of the original container, so all in all three identical container – the original one plus two copies – lie somewhere in memory.

3.3.2 Three kinds of ownership

How can we do better? The most radical solution in terms of efficiency – but also in the sense that it is the unsafest one – would be a view that just stores a

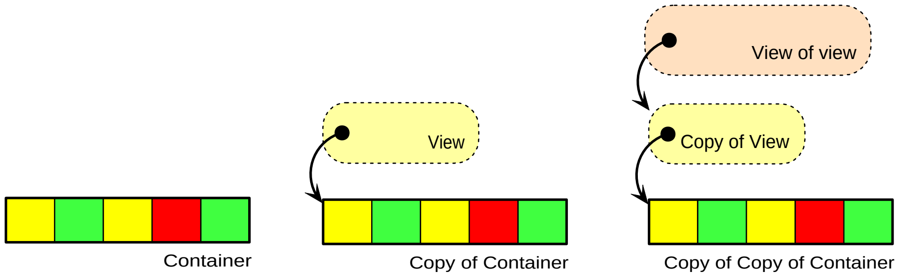

FIGURE 3.1. View and view of a view in case of unique ownership.

pointer to the underlying container instead of the container itself. LISTING 3.3 shows a naive approach to "optimize" things.

```
template<class ContainerT> class my_view
{
  // The constructor.
  my_view( const Container& theData ): ptr( &theData ) {}

  //...
private:
  ContainerT* ptr;
};
```

LISTING 3.3. Code skeleton of another naive view implementation.

Essentially, here we just bend pointers to other locations and need not worry about anything being copied. If we did again construct a view of a view as in LISTING 3.2, just the original container (plus two pointers to it) would be in memory. This is illustrated in FIGURE 3.2.

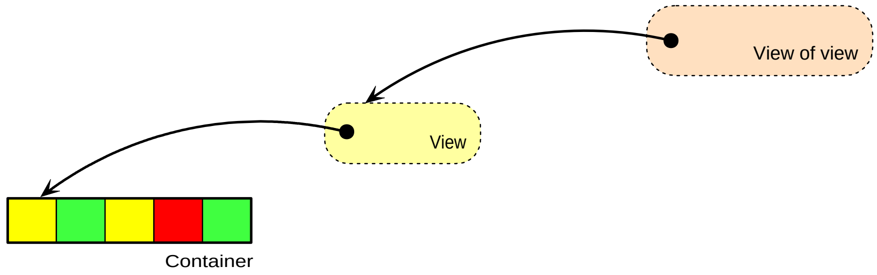

FIGURE 3.2. View and view of a view in case of external ownership.

We need to worry about others issues, however. For instance, assume that somebody writes a function returning a view of a container, say, a vector of **int**'s (LISTING 3.4).

```
my_view< vector<int> > make_a_view()
{
  vector<int> a; // Local w.r.t. function make_a_view().

  //...
  return my_view< vector<int> >( a );
}
```

LISTING 3.4. Good-bye, data!

Whoops! We just created a view containing a dangling reference, that is, a view that references a vector which no longer existing. We traded efficiency for security, which is a bad deal in this case.

One further trial – perhaps the best solution is a compromise between security and performance? Such a compromise – a solution somewhere in between the two previous ones – is offered by using the shared_ptr class of Boost (LISTING 3.5).

```
template<class ContainerT> class my_view
{
  // The constructor.
  my_view( const ContainerT& theData ): ptr( new ContainerT(theData) ) {}

  // The copy constructor.
  my_view( const my_view& other ): ptr( other.ptr ) {}

  //...
private:
  boost::shared_ptr<ContainerT> ptr;
};
```

LISTING 3.5. Code skeleton of a view implementation with shared pointers.

The compromise is that upon construction of a view – for instance, when we create a in line 3 of LISTING 3.2 – a copy of the original container has to be made. However, if then at line 4 b is constructed, we benefit from the copy constructor of shared_ptr. It does *not* copy anything, but just incremented its usage counter, as shown in FIGURE 3.3.

All in all, we came up with three different models how a view can “contain” its underlying STL container:

a) unique: The view contains its own copy of the underlying container. Slow and safe.

b) shared: The view contains its own copy of the underlying container in the form of a reference-counted pointer. Faster and still safe.

c) external: The view has just a pointer which points to the original container. Fast, but unsafe.

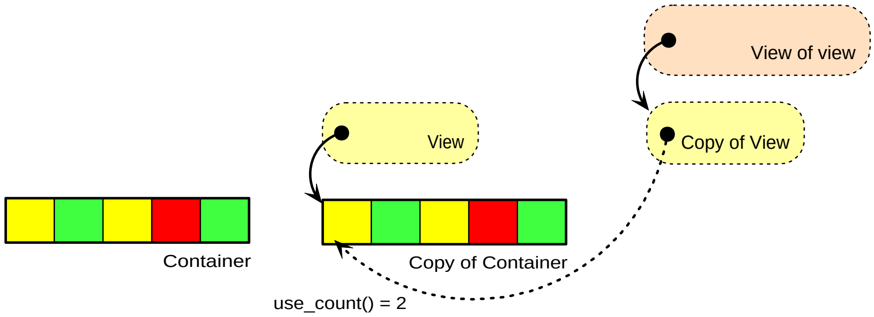

FIGURE 3.3. View and view of a view in case of shared ownership.

3.3.3 Ownership as a policy

Each of the three naive implementations presented above would force a user of the view library to use one specific ownership model. You do not want the ownership model to be hard-coded into the view code, do you? Instead, we want to allow each view type to be *configured* with a ownership policy.

In order to do that, it is necessary that not only the container type, but also the ownership policy is passed as a template parameter to the view class. A quick hack using a template (the container type) and a template parameter (the ownership policy) is

```
template< class ContainerT,
          class OwnershipP = ownership::shared<ContainerT> >
class my_view
{
  //...
private:
  OwnershipP data; // Delegates to ownership.
};
```

Great. Now you can write things like

```
my_view< vector<int>, ownership::unique< vector<int> > a;
```

and you might soon get tired of this, because it not only requires a lot of typing, but it also allows you to write things like

```
my_view< vector<int>, ownership::unique< list<double> > a;
```

which will lead to complete confusion. Besides, this solution does not scale well if we consider views which take *more* than one container as their argument.

Of course, it might be possible to skip the idea of ownership policy at all, or, perhaps, to construct some ingenious solution involving named template parameters. There is a simpler solution, however.

The important observation is that an ownership policy does not exist on its own; it is necessary to put together what belongs together. If only we could write

```
my_view< pair<ContainerT, OwnershipP> >
```

or

```
my_view< OwnershipP<ContainerT> >
```

or something similar! All that is needed is a mechanism to extract both components, ownership policy and container type, back out of the template expressions. Is there such a possibility? Yes, there is, at least if your compiler supports partial template specialization.

The general case, as shown in Listing 3.6, is necessary to define the default policy in case that the user does not explicitly specify it.

```
template<class T> struct wrap
{
  typedef shared<T> type;   // Ownership is "shared" by default
  typedef T         domain;
};
```

Listing 3.6. The ownership wrapper default case ...

The three template specializations then are used if one of the three possibilities is stated explicitly.

```
template<class T> struct wrap< unique<T> >
{
  typedef unique<T> type;
  typedef T         domain;
};

template<class T> struct wrap< shared<T> >
{
  typedef shared<T> type;
  typedef T         domain;
};

template<class T> struct wrap< external<T> >
{
  typedef external<T> type;
  typedef T           domain;
};
```

Listing 3.7. ... and its three specializations.

Now you are able to write

```
my_view< ownership::unique< vector<int> > a;
```

which is a little bit shorter and much less error prone. Voil! Instead of two template parameters, there remains only one (containing both). The view will extract the necessary types out of the ownership wrapper.

```
template<class ContainerT> class my_view
{
  typedef ownership::wrap<ContainerT>::domain domain_type;

  my_view( const domain_type& theData ): data( theData ) { }

  //...
private:
  ownership::wrap<ContainerT>::type data;
};
```

LISTING 3.8. Extraction of necessary types out of the wrapper.

3.4 Some other thoughts

3.4.1 Be aware of what “iterator equality” means

Historically, the iterators of C++ “are a generalization of pointers” [STL]. The syntax and semantics of working with iterators – incrementing, dereferencing, (pointer) arithmetic and so on – closely resembles plain old C pointer handling.

Back in the old days, it was quite clear what “equality” meant: two pointers were considered equal if they pointed to the same location in memory. Naturally, dereferencing two “equal” pointers in the above sense would yield the same values. In short, the following rules were valid:

- Two pointers are equal, i.e. `p1 == p2`, if and only if they point to the same location.
- Both conditions above imply that the dereferenced values are equal, i.e. `*p1 == *p2`.

Since then, things have gotten more complicated than that.

First of all, if two iterators point to the same location, it is no longer guaranteed that dereferencing them results in the same return value. Just consider two transform iterators which iterate over the same range, but have different transform functions, as shown in FIGURE 3.4.

Should we still regard these two iterators as equal? Definitely not! So from the fact that two iterators point to the same location we can no longer conclude that they are equal. Just the converse does still hold: If two iterators do *not* point to the same location, they should *not* regarded as being equal.

- `iter1 == iter2` implies that `iter1` and `iter2` point to the same location.
- If `iter1` and `iter2` point to the same location, then this does neither imply that they are equal, nor does it mean that the dereferenced values are equal, i.e. that `*iter1 == *iter2`.

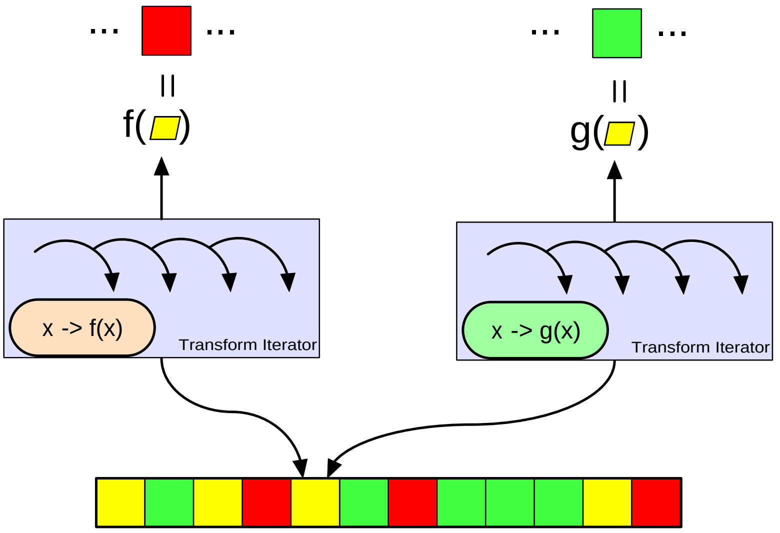

FIGURE 3.4. Two iterators point to the same location, yet yield a different value.

3.4.2 Be aware, part 2: cyclic iterators

Another example concerns a cyclic iterator, which comes in handy to implement a window view. This iterator is intended to re-start iteration at the beginning of a container if it went past-the-end – or to jump back to the last element if it has been at the beginning and was *decremented*. FIGURE 3.5 shows the situation we have in mind.

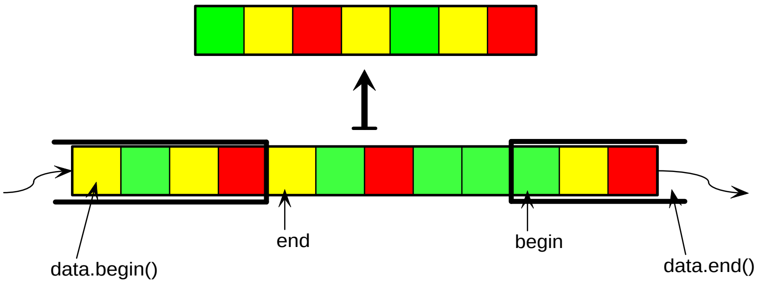

FIGURE 3.5. Cyclic iterator: allows past-the-end iteration over a container.

Implementing a cyclic iterator is not so difficult. All the iterator has to remember are its limits, i.e. the containers' begin and end; *plus*, a counter how often it went "past the end".

Why is this important? Consider a situation as in FIGURE 3.6, written in code as

```
cyclic_iterator iter;
```

```
for( iter = begin; iter != end; ++iter )
  // use *iter
```

We want to iterate over all elements of a container; however, for some strange reason, we want to start at element number 8 (represented by iterator begin), proceed to the end of the container, jump back to the first element, and continue until we reach element 7 (iterator end).

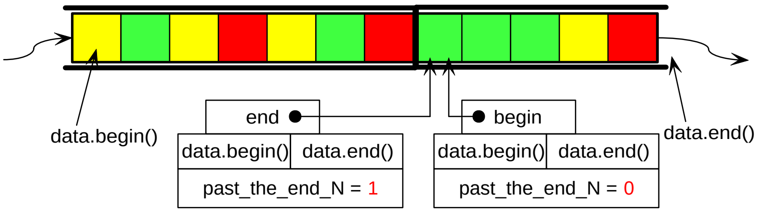

FIGURE 3.6. Beginning and end of a full cycle.

So, the begin iterator points to element number 8; and, due to STL's convention that "end" always means "one past the last", iterator end *also* does point to element number 8! However, we must *not* regard them as equal, otherwise the test iter = end! will stop the iteration immediately.

Here we have the odd situation that two iterators point to the same location, yield the same value when dereferenced, and yet are considered to be *not equal*! We have to conclude:

- Even if two iterators point to the same location and yield the same value upon dereferencing, they are not necessarily equal.

4. TUPLES

> I would rather write programs to help me write programs than write programs.
>
> -Dick Sites[1]

4.1 State of the art

The C++ Standard Template Library [STL] provides (among many other features) the pair<T1,T2> construct – a pair of two data elements of types T1 and T2, respectively. Its only members are these two data elements, called first and second. Construction out of two elements t1 and t2, as well as empty and copy constructors; a make_pair(t1,t2) function for convenience; assignment, equality and 'less than' operators, the later using lexicographic comparison: that is all. This type is merely intended to group two elements together, and, consequently, lacks any further functionality.

Assume, then, that we want to group more than two (or less than two, or precisely two) elements – in short, *any* number of, say, n elements – together: that is what we call an *n-tuple*. In other words, a tuple is a collection of an arbitrary but fixed number of elements, each being of arbitrary but fixed type. Here, "fixed" means "known at compile time".

If size and type were just arbitrary, but not fixed, the means of implementing tuples would be polymorphic types and dynamic lists. For instance, in Java one might use a Vector of elements of type Object to do the job. As flexible as this seems, any type information is lost.

However, since everything really is "fixed at compile time", it is reasonable to let the compiler know it. To do this, the C++ language offers static concepts such as templates. The advantage of this approach is strict type checking, as well as a possible performance improvement.

4.1.1 Boost's tuple implementation

The Boost collection of libraries already contains a tuple implementation written by Jaakko Järvi (see also [Jär99b, Jär99a]). In Boost.Tuple, tuples are based on type or "cons" lists. A type such as

[1] Dick Sites, *Proving that Computer Programs Terminate Cleanly*, Ph.D. thesis, Stanford University, 1974. Quote taken from [CE00, p. 332].

```
tuple<int,double,foo>
```

inherits from

```
cons<int, cons<double, cons<foo,null_type> > >
```

This code snippet looks very much like a LISP list, with the difference that the lists' elements are *types*. `null_type` is an empty type which marks the end of the list.

This structure gives rise to a recursive, "LISP-like" style of programming. For instance, if one likes to retrieve the `N`th element out of the tuple, this is done recursively with the help of class `get_class`. Listing 4.1 shows the general case for `N` being not equal to zero. The recursion stops when `N` gets zero. In that case, the specialization of `get_class` is taken, as shown in Listing 4.2.

```
template< int N >
struct get_class {
  template<class RET, class HT, class TT >
  inline static RET get(const cons<HT, TT>& t)
  {
    return get_class<N-1>::BOOST_NESTED_TEMPLATE get<RET>(t.tail);
  }
  //...
};
```

Listing 4.1. To access an element, go down step by step ...

```
template<>
struct get_class<0> {
  template<class RET, class HT, class TT>
  inline static RET get(const cons<HT, TT>& t)
  {
    return t.head;
  }
  //...
};
```

Listing 4.2. ... until you reach the end. Taken from Boost 1.29.0

All these nested statements are *compile-time* statements. This means that they are resolved at compile-time and are expected to add no run-time overhead (but perhaps compile-time overhead) to the code. For further details, see [Jär99b].

4.1.2 Basic tuple requirements

The intention of this chapter is to present an alternative tuple implementation. Before going into the details, it is reasonable to state which functionality any tuple library should provide. What follows is some sort of a "wish list" – wishes which will become true in a short time –, plus (pseudo-)code snippets:

1. *Tuple construction:* First and fore-most, there has to be some mechanism to create tuples from given data. One can think of several different kinds of construction:

   (a) *Explicit construction:*

   ```
   tuple<int,double,foo> u( 42,3.14,foo("bar") );
   ```

   (b) *Copy construction:*

   ```
   tuple<long,float,foo> w( u );
   ```

   (c) *"Head" plus "tail" construction:*

   ```
   tuple<double,foo> y( 3.14,foo("bar") ); // the tail
   tuple<int,double,foo> z( 42,y ); // head + tail
   ```

2. *Element access:* Certainly we also want methods to get the data back out from the tuple. For a tuple of n elements, we need functions get0 up to get*k*, *k*=n−1. These functions should allow both read and write access whenever possible; if not possible – for instance, in the case of a **const** type – it should allow read access only, and issue a compiler error if one tries to write to it.

   (a) *Access via member function:*

   ```
   int i = u.get0();
   u.get1() = 2.78;
   ```

   (b) *Access via global function:*

   ```
   double d = get1(u);
   get2(u).doo("ogg"); // calls foo method
   ```

   (c) *"Head" plus "tail" access:*

   ```
   double pi = u.tail().head();
   ++u.head();
   ```

3. *Assignment and swapping:* We want to assign and swap tuples in the expected way. Assignment, as well as copy construction, should also perform some kind of type cast, as in

   ```
   tuple<int,double> u( 42,3.14 );
   tuple<long,float> v;
   v = u; // ok, cast types
   ```

4. *make_tuple and tie:* For convenience, we want two helper functions to simplify construction of tuples and extracting data back out into individual variables:

   ```
   tuple<int,double,foo> u = make_tuple( 42,3.14,foo("bar") );

   int i; double d; foo f;
   tie( i, ignore, f ) = u;
     // now i == 42 and f == foo("bar").
   ```

5. *Relational operators:* We need to test tuples for equality and inequality. Additionally, testing for less than etc. w.r.t. lexicographical ordering is useful sometimes:

   ```
   tuple<int,double,foo> u( 42,2.78,foo("bar") );
   tuple<long,float,foo> w( 42,3.14,foo("bar") );

   if( u == w ) // element-wise equality
     // do something

   if( u < w ) // lexicographical ordering
     // do something different
   ```

6. *Helpers for functions:* Say there is a function that expects *three* separate arguments; however, we would like to pass *one* 3-tuple to it. For such situations, a helper function which maps one calling form to the other comes in handy:

   ```
   bool is_fooable( int,double,foo ); // a function taking three args

   tuple<int,double,foo> u( 42,2.78,foo("bar") ); // the argument

   pointer_to_function<bool,int,double,foo> fooable( is_fooable );

   if( fooable(u) ) // calls is_fooable( get0(u),get1(u),get2(u) )
   ```

Of course, as always, everything should be extremely fast, and nicely documented, and highly portable, and ... Just wait. There is yet another wish.

4.1.3 Additional functionality

Boost.Tuple does provide (almost) all of the above items, so why rewrite it? The motivation to reimplement the tuple type was that I wanted to group *containers* and *iterators* together in a tuple in the same manner as “plain old data types”. This is necessary, for instance, to generalize the implementation of a zip_view [PW00].

The intended meaning of a container tuple type is straight-forward: A tuple of, say, three containers (always in the sense of STL containers such as std::vector, std::list etc.) containing **int**s, **double**s,and foos, respectively, should be “the same” as a container which contains tuple<**int**,**double**,foo>s as elements. Of course, we can not expect a container tuple type to provide the same rich variety of methods as STL containers in general do. Especially, it would be difficult to implement methods to insert tuples of elements into or remove tuples of elements out of a tuple of containers. Functionality that can be implemented anyway (often in a straight-forward way) include the functions empty() and size() and the iterator mechanism associated to a standard container – that is, methods begin() and end().

That, in turn, requires a tuple of *iterators* together with its methods such as increment, decrement, and dereference. LISTING 4.3 shows a sketch of the intended usage of container and iterator tuples. Another nice feature would be

```
vector<int> u; vector<double> v; vector<foo> w;
// fill u, v, and w with data

typedef container_tuple< vector<int>,vector<double>,vector<foo> >
        my_vectors;

my_vectors t( u,v,w );
my_vectors::iterator it;

for( it = t.begin(); it != t.end(); ++it )
{
  // (*it) is of type tuple<int,double,foo>:
  int i = get0( *it );
  it->get1() = 3.14 * i;
}
```

Listing 4.3. A tuple of containers appears as a container of tuples.

to overload the meaning of **operator**[] to allow access to the individual elements of a tuple of containers. This is not always appropriate; when it is, however, it makes sense to provide it. See Listing 4.4 for an example.

```
// u, v, and w as above
my_vectors t( u,v,w );

tuple<int,int,int> index( 1,4,3 );
t[ index ] = make_tuple( 42,3.14,foo("bar") );
```

Listing 4.4. Access via indexing is extended.

4.1.4 Tuples and beyond

Which brings us to the question: where to put all this methods, container and iterator functionality? Consider Listing 4.5 – it is tempting to add these "common" methods, such as increment, addition, minimum, and so on, to the basic tuple type. They just occur so frequently, and it seems to be easy to implement them.

Yet, I decided against stuffing too much functionality into the base tuple type. Instead, the base idea is to keep the base type simple and enrich the derived types with more functionality.

- The base type `tuple` provides all the basic functionality as listed in Subsection 4.1.2 – but nothing else.
- Derived types add more functionality; they may impose additional requirements to their elements' types.
  - `container_tuple`: assumes that elements are STL containers

```
tuple<int,double> u( 42, 3.14 );
tuple<long,float> v( 77, 2.78 );

// element-wise addition
tuple<long,double> w = u + v; // gives ( 119, 5.92 )

// vector product
double p = u * v; // gives 42 * 3.14 + 77 * 2.78 = 345.94

// element-wise minimum
tuple<long,double> m = min( u, v ); // gives ( 42, 2.78 )

// write to standard output
std::cout << m;
```

Listing 4.5. Should we add all these functions?

- iterator_tuple : assumes that elements are STL iterators
- math_tuple: provides functionality for "math-like" tuples, i.e. operations such as addition, multiplication and the like.
- (to be continued)

4.2 A new approach

The approach taken here is in fact very, very simple. It closely resembles an approach "as if written by hand". Actually, we could take the (hand-written) implementation of std :: pair<T1,T2> as a starting point.

As already pointed out, all type informations are already fixed at compile time. Therefore static concepts are preferable: we will implement the tuple type(s) by means of templates. Our starting point is a general tuple template which is *empty* – it just serves as a placeholder.

Since we would like to have tuples for a various number of elements (remember: "arbitrary, but fixed"), we have to implement a structure for each N. So, subsequent tuple structs are implemented as a template specialization from the general placeholder. These ideas are outlined in Listing 4.6.

In fact, we can already draw some important conclusions from the structure shown in Listing 4.6. In comparison to Boost.Tuple, the new approach as presented here has the following characteristics:

1. Our tuples are flat, not nested. This is not a benefit *per se*; however, flat structures might result in simpler, and therefore more efficient, code.

2. The code of Listing 4.6 requires partial template specialization. This means that some compilers are not (yet) able to compile such tuples.

3. There are lots and lots of code to write. In some sense, our "brute-force approach" lacks the elegance of tricky template meta-programming as shown in Boost.Tuple.

```
// The empty type.
struct null_type {};

// The general tuple template. Just a placeholder to ``specialize from''.
 template< class T0 = null_type , class T1 = null_type , ... >
 struct tuple
 { };


// Template specialization for N = 2:
template< class T0 , class T1 >
struct tuple< T0 , T1 , null_type , null_type , null_type , ... >
{
  T0 m0; T1 m1;
};

// Template specialization for N = 3:
template< class T0 , class T1 , class T2 >
struct tuple< T0 , T1 , T2 , null_type , null_type , ... >
{
  T0 m0; T1 m1; T2 m2;
};

// ... and so on ...
```

Listing 4.6. Code skeleton of hand-written tuple implementation.

The next step is to add some basic functionality such as constructors and get methods. This can still be done in a straight-forward way, without too much thinking. The general rule of thumb is to break down each method to its element-wise components. As a bonus, we integrate automatic element-wise type conversion into the copy constructor.

The code of Listing 4.7 will work fine until, for some reason, you start to instantiate objects of type `tuple<int&,double&>`; in such a case, your favourite compiler might tell you that it does not like references of references. As these problems are already addressed in the Boost.Tuple library, I will not go into details here.

4.2.1 Code generation

By now, it is obvious that the tuple code grows and grows; each modification has to be written (and maintained thereafter) for each N-tuple. "Do it yourself" might be tedious and error-prone – it pays off to remember the advice of the beginning: instead of writing tuple code, write code that writes tuple code.

Although there are a couple of suitable scripting languages which are able to generate code in a systematic manner, we will stay within the scope of the C language. Indeed, each C compiler *is able* to generate code automatically: by means of macros fed to the preprocessor.

Conventionally, the C preprocessor is seen as "relatively unsophisticated" [Str97],

```
// Template specialization for N = 3:
template< class T0 , class T1 , class T2 >
struct tuple< T0 , T1 , T2 , null_type , null_type , ... >
{
  // Constructs tuple out of elements.
  tuple( T0 theM0 , T1 theM1 , T2 theM2 )
    : m0(theM0) , m1(theM1) , m2(theM2)
  { }

  // Copy constructor. Does element-wise type cast.
  template< class S0 , class S1 , class S2 >
  tuple( const tuple< S0 , S1 , S2 >& rhs )
    : m0( rhs.m0 ) , m1( rhs.m1 ) , m2( rhs.m2 )
  { }

  const T0& get0() const { return m0; }
        T0& get0()       { return m0; }

  // same for N = 1,2.

  T0 m0; T1 m1; T2 m2;
};
```

Listing 4.7. Constructors and get-methods added to the skeleton.

and its use is regarded to be dangerous. Yet, the preprocessor is suitable to generate repetitive code structures; with the help of the Boost.Preprocessor library, we can even formulate a general tuple type macro. That's what I would like to call the "Tupple" library: tuples from the preprocessor.

For instance, assume that we want to "automatize" the generation of the tuple members. That is, the code line

```
T0 m0; T1 m1; T2 m2; T3 m3; T4 m4;
```

(assuming N=5, of course) should be generated rather than written manually.

This turns out to be quite easy: all there is to do is to define a single macro

```
#define MEMBER(z,k,_) T##k m##k;
```

Note that the macro operator "##" stands for string concatenation. That is, a call to MEMBER(z,5,_) would expand to the text "T5 m5;". The first and third argument of the macro[2] are ignored.

Given that, just include the proper header files of Boost.Preprocessor and start compiling; the code snippet

```
BOOST_PP_REPEAT(5,MEMBER,_)
```

then generates the line we want.

[2] All examples are done with Boost.Preprocessor library of Boost version 1.29.0, which was a major revision. For instance, only since version 1.29.0 three macro parameters are required; in former versions, it was sufficient to write #define MEMBER(k,_) T##k m##k;

```
// define a reasonable number MAX_N as maximum for template parameters

// Template specialization for N = 3:
template< enumerate "class Tk" for k = 0,1,2 >
struct tuple< enumerate "Tk" for k = 0,1,2
              here comes a comma "," if k is not zero
              "null_type" for (MAX_N - k) times >
{
  tuple( enumerate "Tk theMk" for k = 0,1,2 )
     : enumerate "mk(theMk)" for k = 0,1,2
  { }

  // ...

  repeat "Tk mk;" for k = 0,1,2
};
```

Listing 4.8. Abstract structure of tuple template with N=3.

In principle, this can be extended to more complex expressions necessary for tuple code generation. For instance, reconsider Listing 4.7; if we would like to extract the underlying structure of the code in order to see how to generate it, the result might look like Listing 4.8.

The only task left is to fill out this structure, that is, to give a meaning to commands like "enumerate *text* for k=0,1,...,N". Given the capabilities of the the Boost.Preprocessor library, this occurs to be almost trivial. The only requisite to do so is to define the necessary macros. As a result, we can define another macro, say STRUCT_TUPLE(k), which is capable of generating the source code of **struct** tuple[3] for each non-negative k. A call to STRUCT_TUPLE(3) would then produce the whole code of a tuple with three elements. Listing 4.9 has the details.

The same code generating strategy applies to types which are derived from tuple, such as container_tuple and iterator_tuple as described in Subsetion 4.1.4.

For instance, the increment operator

```
self_type& operator++() {
  ++m0; ++m1; ++m2; ++m3; ++m4;
  return *this;
}
```

of an iterator_tuple of size 5 can easily be generated by

```
#define INC(z,k,_) ++m##k;
//...
self_type& operator++() { BOOST_PP_REPEAT(k,INC,_) return *this; }
```

– likewise, the empty() function of a container_tuple , which I implemented as

[3] Obviously, such a macro will be quite long – definitely longer than one line. Keep in mind that the backslash character '\' is used to adjoin subsequent lines in macro definitions.

```
#define MAX_N 10

#define NULLTYPES(z,k,_) null_type

#define CTORARG(z,k,_) T##k theM##k
#define INITCTOR(z,k,_)  m##k(theM##k)
#define MEMBER(z,k,_) T##k m##k;
//...

// Template specialization for all N.
#define STRUCT_TUPLE(k)                                                \
template<BOOST_PP_ENUM_PARAMS(k,class T)>                              \
struct tuple< BOOST_PP_ENUM_PARAMS(k,T)}                               \
              BOOST_PP_COMMA_IF(k)                                     \
              BOOST_PP_ENUM( BOOST_PP_SUB(MAX_N,k) ,NULLTYPES,_) >     \
{                                                                      \
  tuple( BOOST_PP_ENUM(k,CTORARG,_) )                                  \
  : BOOST_PP_ENUM(k,INITCTOR,_)                                        \
  { }                                                                  \
                                                                       \
  BOOST_PP_REPEAT(k,MEMBER,_)                                          \
};

STRUCT_TUPLE(1)  // generate code for N=1
STRUCT_TUPLE(2)  // generate code for N=2
STRUCT_TUPLE(3)  // generate code for N=3
// etc.
```

Listing 4.9. Abstract tuple template filled with macro definitions.

```
tuple< bool , bool , bool> empty() const {
  return tuple< bool , bool , bool>( m0.empty() , m1.empty() , m2.empty() );
}
```

is the result of yet another macro definition, namely of

```
#define NTIMES(z,k,arg) arg
#define EMPTY(z,k,_) m##k.empty()

TUPLE(k)<BOOST_PP_ENUM(k,NTIMES,bool)> empty() const {                  \
  return TUPLE(k)<BOOST_PP_ENUM(k,NTIMES,bool)>( BOOST_PP_ENUM(k,EMPTY,_) ); \
}
```

Suffice it to say that with the of the Boost.Preprocessor library, we can write one macro to generate all the tuple code, one macro for tuples of iterators, and so on; in fact, a very limited subset of Boost.Preprocessor (mainly enumeration and repetition) is necessary.

4.2.2 Sequence of use

In general, it is possible to use the tuple generating macro code in two different ways:

- Include the file containing the macro definitions. The code of the tuple structure will be generated each time the header file is included.
- Let the macro definition generate a separate file. Include only this file which contains the generated code (but no macros any more).

As the generated code grows longer and longer, the second possibility gets more preferable. In the last subsection I claimed that the STRUCT_TUPLE(k) macro as defined in Listing 4.9 is capable of generating the complete tuple type code for all non-negative k. What is actually generated is this:

```
template< class T0 , class T1 , class T2> struct tuple < T0 , T1 , T2 , null_type , null_type , null_
```

– everything in one line. The preprocessor does not care much about indent and a nice layout. If we would like to obtain a separate tuple library file, we have yet to care about formatting the generated text along the following rules:

- Increase the indent after an opening curly bracket "{", and decrease it after a closing one "}".
- Place line breaks at proper places – for instance, after ";", and after curly brackets.
- Take care that each preprocessor directive such as #include <somefile> appears on a new line.

5. EXAMPLES

> In theory there is no difference between theory and practice. In practice there is.
>
> - Yogi Berra

5.1 String parsing

As a first example, lets look at the processing of strings (which can be "viewed" as a container of characters). Suppose we want to split up a string such as

```
One sentence with five words.
```

into its five single words `One`, `sentence`, `with`, `five`, and `words` – the later without the period. Again, a picture can help to understand our goal – see FIGURE 5.1. Note that this picture in some sense looks like the "inverse" of the chain view picture.

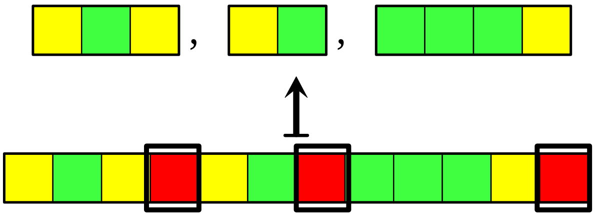

FIGURE 5.1. Parsing does not modify the containers' contents, but modifies the structure.

The first trial I undertook some time ago looked like LISTING 5.1.

Wait, what's happening here? We use the transform view to iterate over the string; each time the transform views' function returns true – which means that the iterator points to a blank in between words – we send the string formed from between `b` and `e` to the standard output. That's not very elegant, and in fact we can go much further than that.

To do so, lets write down what parsing means. Given a string, which we treat as a container of characters, here's the strategy:

```
string s( "One sentence with five words." );

typedef transform_view< string, boost::function1<bool,char> >
        TrueIffBlankView;
TrueIffBlankView view( s, bind2nd( equal_to<char>(), ' ' ) );

TrueIffBlankView::const_iterator b = view.begin();
TrueIffBlankView::const_iterator e;

while( ( e = find( b, view.end(), true ) ) != view.end() )
{
  cout << string( s.begin() + (b-view.begin()),
                  s.begin() + (e-view.begin()) ) << endl;
  b = e + 1;
}
```

Listing 5.1. Writes the five words to the screen.

1. Place breaking marks at certain locations – for instance, blanks, commas, periods, etc. That's what a filter view can do: given a "is a character indicating a break"-predicate, it only returns (pointers to) those characters.
2. Form new strings which hold exactly the text between two breaking marks. That might be the task of a transform view which is initialized with some string generating function. Since a transform view can only hold a unary function, we have to add another view:
3. Pair together two consecutive (pointers to) breaking marks. Each such pair describes the range of a string, stretching from the pairs' first member up to the pairs' second member. Pairing together might be done with the help of a neighbour view.

Summarizing, we have got to deal with three different views: First, a filter view filters out only those characters which indicate a breaking location. Second, a neighbour view pairs together consecutive such locations together into ranges. Third, a transform view takes such ranges and creates string out of them.

```
struct isBreakingCharacter
  : public std::unary_function< char, bool >
{
  bool operator()( char ch ) const
  {
    return( isalnum( ch ) == 0 );
  }
};
```

Listing 5.2. Break if character is not alpha-numeric.

Assuming that we have two functions (or rather function objects), namely, isBreakingCharacter() which tells us whether we should break at a certain character or not (compare Listing 5.2, and constructString() which constructs strings out of ranges, we can define the exact view types. Note that LISTING 5.3 shows very clearly how one view is stacked upon the other.

```
typedef filter_view< string, isBreakingCharacter >
        OnlyCharsAtBreakView;

typedef neighbour_view< OnlyCharsAtBreakView, 2 >
        PairTogetherView;

typedef transform_view< PairTogetherView, constructString >
        ConstructStringsView;
```

LISTING 5.3. Definition of the three views.

Using these type definitions, writing the working code is relatively straightforward. Unfortunately, there's just one further detail which has to be observed: the first breaking character appears somewhere after the beginning of the string. In our case, the first blank is *after* the word "`One`". Consequently, the first pair of consecutive characters subsumes the blank after `One` and that after `sentence`; which means that the first constructed string is "`sentence`" – we missed the first word! As a workaround, lets insert a blank at the front of the sentence; finally, we can parse the string as shown in LISTING 5.4.

```
  std::string s( "One sentence with five words." );

  OnlyCharsAtBreakView onlyAtBreak( s );

  // HACK Inserts break point at front of s to get the first word as well:
  onlyAtBreak.domain().insert( (std::string::size_type)0, 1, ' ' );

  PairTogetherView::difference_type consecutivePairs( 0, 1 );
  PairTogetherView pairTogether( onlyAtBreak, consecutivePairs );

  ConstructStringsView result( pairTogether );
```

LISTING 5.4. The final code.

It is noteworthy – and a little bit disappointing – that even this relatively simple task requires not only to stack three different views upon each other, but also some non-intuitive hacks. Partly, this is so because views only allow local operations; for algorithms which require non-local data – and parsing is such an operation –, we have to apply certain tricks.

On the other hand, this examples also demonstrates the "pros" of using views: for instance, up to the last line, where the final view `result` is constructed, we did just that: constructing views. We did *not* have any other computations due to lazy evaluation.

Out of curiosity, I'd like to advance this example a little bit further. Sometimes it is desirable to keep a certain part of the text "as is" although it contains breaking characters. A simple possibility to tell the algorithm that it should left this part unchanged is to surround this part with so called *escape characters* – typically, quotes, brackets and the like. Suppose we have got

```
A sentence "for demonstration purposes only" with nine words.
```

This sentence should be split up into *six* parts: namely, `A`, and `sentence`; the fragment `"for demonstration purposes only"` should be kept as a whole; then, `with`, `nine`, and `words`.

One solution might be to extend function `isBreakingCharacter()` as shown in Listing 5.5: an internal state `insideQuotes` is added to the function object which keeps records of whether it is currently inside or outside a matching pair of quotes.

One possible drawback of this straight-forward implementation is that it assumes that the string is parsed from left to right. This is a general danger when working with functions that have an internal state. Note however that a filter iterator already is forward only, so this assumption does not impose any further restriction.

```
struct isBreakingCharacter
  : public std::unary_function< char, bool >
{
  isBreakingCharacter()
    : insideQuotes( false )
  { }

  bool operator()( char ch )
  {
    if( ch == '\"' )
    {
      insideQuotes = !insideQuotes;
      return false;
    }
    else
      return( !insideQuotes && ( isalnum( ch ) == 0 ) );
  }

private:
  bool insideQuotes;
};
```

Listing 5.5. Breaks at non-alphanumeric characters *outside* quotes.

5.2 Signal processing

5.2.1 Sampling

To be able to apply signal processing algorithms, we first need a *discrete* signal. Hence, the starting example to demonstrate the application of views in the

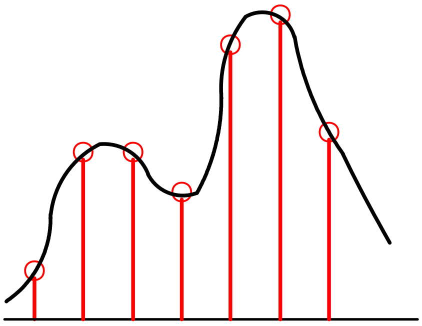

FIGURE 5.2. Continuous signal is sampled to obtain a discrete one.

signal processing domain is to down-sample a continuous signal into a discrete one.

Suppose we have a formula which describes the given signal. Sure enough, we can put this generating function into a function_view and already get a discrete signal with step width 1. To sample the signal with another sampling rates, say, with step width 2, we could either

1. manually iterate through the function_view with step 2,
2. use a step_iterator with step width 2 to iterate through the function_view, or
3. wrap a permutation_view with an appropriate re-indexing scheme around the function_view.

The later case is shown in LISTING 5.6. Note that due to the "laziness" of views, the generating function is not called until the value is actually fetched. Thus, either the sampled view itself can be used, or, as shown in LISTING 5.6, we decide to copy the sampled signal into another container such as a vector.

5.2.2 Windowing

One of the most important tools to analyse a (discrete and periodic) signal is the Discrete Fourier Transform (DFT), which converts the given signal into its frequency domain representation. In order to avoid some unwanted effects such as spectral leakage [Smi97], the signal is often multiplied with a window function before calculating its DFT.

There exists a multitude of different window functions; one of the most popular is the *Hamming window*. Given a discrete signal of length N, the Hamming window is described by the formula

$$w[k] = 0.54 - 0.46 \cos \frac{2\pi k}{N+1} \quad \text{for } k = 0, \ldots, N.$$

```
typedef boost::view::function_view< GenerateSignal > SignalFunction;
typedef boost::view::permutation_view<
          SignalFunction, std::vector<int> > SamplingView;

// Samples in reverse order, and only every second value.
std::vector<int> indices;
for( int i = 0; i < 64; ++i )
  { indices.push_back( 2*(63-i) ); }

SignalFunction signalF( 0, 64 );
SamplingView sampledF( signalF, indices );

std::vector<double> signal( sampledF.size() );
std::copy( sampledF.begin(), sampledF.end(), signal.begin() );
```

Listing 5.6. Sampling a continuous signal.

We might now implement this operation in terms of the data/view model: first of all, the signal itself is assumed to be stored in a container, such as a STL vector. The Hamming window can be represented by a function view. The element-wise multiplication of the signal with this view requires a transform view with multiplication as its transformation function.

```
typedef std::vector<double> Signal;
typedef boost::view::function_view<HammingWindow> HammingView;

typedef boost::tupple::container_tuple<Signal, HammingView> SignalPairs;
typedef boost::view::transform_view<SignalPairs, Multiplication>
  WindowedSignal;
```

Listing 5.7. Signal container and window view combined.

Since this is a binary function, and since a transform view can only operate on a single container with a unary function, another in-between mechanism is necessary which glues together these two containers (or rather, the container and the view) to form another container holding pairs of elements. Thus, once again, an intermediate view which pairs elements together is necessary; this time, however, pairs do not consist of elements from the same container (as it was the case in the last example), but of corresponding elements of two containers of the same size. As outlined in Chapter 4, a container_tuple of size 2 does exactly this, and it does not make any difference that one of its arguments is not a container, but a function view.

Using the two functions (or rather function objects) HammingWindow which calculates the Hamming function, and Multiplication which multiplies the two coefficients of a pair, Listing 5.7 shows all types necessary to accomplish our task. Assuming that $N = 64$, these types can then be used to construct the necessary views, as shown in Listing 5.8.

```
Signal signal;
// Fill signal with data.

HammingView hamming( 0, 64, HammingWindow( 64 ) );
SignalPairs sigPairs( signal, hamming );
WindowedSignal windowed( sigPairs );
```

Listing 5.8. Multiplies a signal with a Hamming window function

5.3 Image iteration

Iteration is inherently a one-dimensional concept. Returning once again to our famous loop,

```
for( int i = 0; i != N; ++i )
  result[i] = function( source[i] );
```

how can we extend that basic mechanism to two- (or more) dimensional data? Using indices and `operator[]`, this seems really quite trivial:

```
for( int j = 0; j != M; ++j )
  for( int i = 0; i != N; ++i )
    result[i][j] = function( source[i][j] );
```

Assuming that an image is stored as a `vector` of `vector` of pixels, i.e. as `vector< vector<PixelType> >` – which is not a common way of storing images – our loop reads like this:

```
vector< vector<PixelType> >::iterator row;
vector<PixelType>::iterator col;

for( row = image.begin(); row != image.end(); ++row )
  for( col = row.begin(); col != row.end(); ++col )
    // uses *col
```

Of course, if the image was stored as one large chunk of data in memory – and that *is* most often the case –, one might as well apply apply one-dimensional iteration. However, structure information is lost that way. For instance, if we wanted to process a rectangular section of the image only, this kind of representation would be rather troublesome.

5.3.1 Two-dimensional iteration

As an example how to implement two-dimensional image iteration, I present the VIGRA image processing software package [Köt]. As described in the manual, and in more detail in [Köt99] and [Köt00], VIGRA employs two-dimensional iteration. As Ullrich Köethe points out this "is not directly possible using operator overloading." Instead, a nested class `ImageIterator` is created which contains the structures to iterate both in horizontal and vertical direction. This allows to iterate in both directions independently, as shown in Listing 5.10.

```
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
};
```

Listing 5.9. Outline of two-dimensional image iterator. From [Köt00].

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

Listing 5.10. Iteration in two dimensions. From [Köt00].

5.3.2 Matrix view

In this presentation, I'd like to develop another approach. Remember that a view can change the appearance of a container. So why not looking for a view which attaches a two-dimensional "look" to a one-dimensional container?! More precisely, we strive for a view which wraps a one-dimensional container and provides some kind of iterator that allows two different types of iteration:

- inner (horizontal) iteration: proceeds from one pixel to the next between a given `begin/end` pair which delimits the current row.
- outer (vertical) iteration: moves the `begin/end` pair from one row to the next.

Such a structure is what I'd like to call a *matrix view*. Figure 5.3 depicts the situation once again. Translated to code, the two nested loops to iterate over the complete image (or over a rectangular part of it) will read something like this:

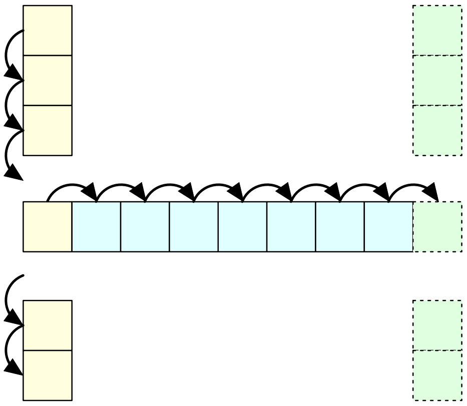

FIGURE 5.3. Iteration over an image: whereas the outer loop iterates over rows (given as begin/end-pairs), the inner loop iterates over pixels.

```
// Outer loop advances from one row to the next
for( MatrixView::iterator row = view.begin(); row != view.end(); ++row )
{
  // Inner loop advances from one pixel to the next
  for( PixelIterator pixel = row.first(); pixel != row.last(); ++pixel )
  {
    // use *pixel
  }
}
```

5.4 Image processing and enhancement

Now that we found a suitable way how to iterate over a two-dimensional image, and therefore provide a way how to treat an image as a view, the next question I want to raise is: which image processing algorithms can be re-formulated in terms of the data/view model?

Definitely, not every algorithm can be transformed. As an example, consider the connected components algorithm, which determines the component a pixel is contained in within a (binary) image. This algorithm obviously requires global image information; since views operate on a local level, this algorithm is not suitable for use with the data/view model.

On the other hand, if we restrict ourselves to "local operations", a re-formulation might be possible and useful. For instance, pixel-wise operations are ideally suited for the use of views; such operations could be, for instance,

- colour-to-colour or colour-to-grey scale conversions
- calculation of the negative image
- brightness, contrast, saturation, or colour enhancements.

Also operations which consists essentially of a re-ordering of pixels are candidates for a re-formulation. Typical simple examples are if one wants to

- rotate an image left or right, or
- flip an image vertically or horizontally.

Many image processing algorithms are "local" in the sense that they only require a small neighbourhood around the current pixel to work on. Examples are

- local filter operations such as blur or sharpen
- most edge detection algorithms.

Another such function which is calculated using a 3 × 3 neighbourhood, and which I'd like to consider in greater detail in the next subsection, is the *discrepancy norm* [BBK96].

One question arises when working with such functions: How to deal with image boundaries? In principle, there are two ways how to proceed at the boundaries where the neighbourhood exceeds the image borders vertically or horizontally:

1. Ignore such pixels completely. For instance, for a 3 × 3 neighbourhood, all pixels in the top-most and bottom-most row and in the left-most and right-most column would be skipped.
2. Wrap the neighbourhood around the image boundaries.

For the later approach, using a cyclic iterator which wraps around containers might be useful, as was discused in SUBSECTION 3.4.2.

5.4.1 Image segmentation with the discrepancy norm

Among the multitude of algorithms to detect edges in images, there are some approaches that utilize fuzzy reasoning. More precisely, for each pixel in the image some suitable classification numbers are calculated; then, these numbers are used as input values to a fuzzy system which contains the rules describing the "edginess" and which decides whether the pixel is an edge pixel or not.

This mechanism can be either supervised, as described in [Ara00], or unsupervised, as is the approach using the discrepancy norm presented in [BBK96]. This approach has the additional advantage that it does not only separate between edge and non-edge pixels, but does a classification for each pixel whether it lies in a "Homogeneous", "Edge", "Halftone", or "Picture" area.

The approach of [BBK96] is based on the discrepancy norm[1].

[1] The definition of the discrepancy norm dates back to 1916 [Wey16]. Its application in pattern recognition was first pointed out in [NW87].

5.4.1. Definition.

The mapping

$$\begin{aligned} \|.\|_D : \mathbb{R}^n &\longrightarrow \mathbb{R}, \\ \vec{x} &\longmapsto \max_{1\leq a\leq b\leq n} \left| \sum_{i=a}^{b} x_i \right| \end{aligned}$$

is called the *discrepancy norm* on $\mathbb{R}^n$.

Since this formula would require $\mathcal{O}(n^2)$ operations to calculate, the following formula is more practical to use:

5.4.2. Theorem.

Let $X_j := \sum_{i=1}^{j} x_i$ denote the partial sums for $1 \leq j \leq n$. Then, for all $\vec{x} \in \mathbb{R}^n$,

$$\|\vec{x}\|_D = \max_{1\leq b\leq n} X_b - \min_{1\leq a\leq n} X_a$$

holds.

Proof.

See [BBK96]. □

Time for some definitions: we work with either grey level or RGB images of dimension $W \times H$, where each of the three colour bands has 1 byte, i.e. 8 bits, of information. More formally, this reads as:

5.4.3. Definition.

A $W \times H$ matrix of the form

$$(v(i,j)),\ i = 0,\ldots,W-1,\ j = 0,\ldots,H-1$$

is called an 8 bit grey level image of width W and height H. Its entries $v(i,j) \in \{0,\ldots,255\}$ are called pixels at (i,j).
A $W \times H$ matrix of the form

$$((r(i,j), g(i,j), b(i,j))),\ i = 0,\ldots,W-1,\ j = 0,\ldots,H-1$$

is called a 24 bit RGB colour image of width W and height H. Its entries

$$p(i,j) := (r(i,j), g(i,j), b(i,j)) \in \{0,\ldots,255\}^3$$

are called RGB pixels at (i,j), where $r(i,j)$ represents the red, $g(i,j)$ the green, and $b(i,j)$ the blue portion of the pixel.

In order to calculate the discrepancy norm for a given 3 × 3 neighbourhood of a pixel, the neighbouring pixels have to be enumerated such that they form a tuple of size eight. This is done as shown in Figure 5.4, that is, we define an enumeration mapping $l_{i,j} : \{1,\ldots,8\} \longrightarrow 0,\ldots,W-1 \times 0,\ldots,H-1$ which

2	3	4
1	☒	5
8	7	6

FIGURE 5.4. Enumeration of pixels within the 3×3 neighbourhood.

maps $1 \mapsto (i, j-1)$, $2 \mapsto (i-1, j-1)$ and so on.

Then we define

$$e(i,j) \quad := \quad \|v(l(.)) - (\bar{v}, \ldots, \bar{v})\|_D$$

for the grey level case, and

$$\begin{aligned} e(i,j) \quad := \quad & \|r(l(.)) - (\bar{r}, \ldots, \bar{r})\|_D \\ & + \|g(l(.)) - (\bar{g}, \ldots, \bar{g})\|_D \\ & + \left\|b(l(.)) - (\bar{b}, \ldots, \bar{b})\right\|_D \end{aligned}$$

for the RGB colour case, where $\bar{v}$, $\bar{r}$, $\bar{g}$, and $\bar{b}$ denote the mean values, i.e. $\bar{v} = \frac{v(l(1))+\cdots+v(l(8))}{8}$ and so on.

In FIGURE 5.5, several different neighbourhoods and the corresponding values of $e(i,j)$ are shown. For the grey level case – which might be easily generalized to the colour case – we might observe the following:

OBSERVATION 1. $e(i,j)$ is zero in a completely homogeneous area.
All eight entries of the neighbourhood being equal means $v(l(1)) = v(l(2)) = \ldots = v(l(8)) = \bar{v}$, which implies $\|v(l(.)) - (\bar{v}, \ldots, \bar{v})\|_D = \|(0, \ldots, 0)\|_D = 0$.

OBSERVATION 2. $e(i,j)$ is relatively low when pixel values alternate between black and white, as is the case in half-tone, "chequerboard-like" areas.
Assume that there is an a such that the pixels at $(i-1, j)$, $(i, j-1)$, $(i+1, j)$ and $(i, j+1)$ — the "white" pixels of the chequerboard—have a value of $\bar{v} + a$, whereas the black pixels have a value of $\bar{v} - a$. Then $e(i,j) = \|(\bar{v}+a, \bar{v}-a, \bar{v}+a, \bar{v}-a, \ldots) - (\bar{v}, \ldots, \bar{v})\|_D$; since $\max X_b = \max(a, a-a, a-a+a, \ldots) = a$ and $\min X_b = \min(a, a-a, a-a+a, \ldots) = 0$, their sum is $e(i,j) = a$.

OBSERVATION 3. $e(i,j)$ has its maximum if the neighbourhood has a sequence of black pixels followed by another sequence of white pixels, as shown in the right-most example of FIGURE 5.5.
Assume that there is an a such that the first four pixels have a value of $\bar{v} - a$, the second consecutive four one of $\bar{v}+a$. Then $e(i,j) = \|(a, a, a, a, -a, -a, -a, -a)\|_D$; since $\max X_b = 4a$ and $\min X_b = 0$, $e(i,j) = 4a$.

Out of these facts we can conclude that $e(i,j)$ indeed serves as an indicator to which degree the pixel is lying at or near an edge. Additionally, as tests in [BBK96] indicate, it is more robust w.r.t. noise than other conventional edge detectors.

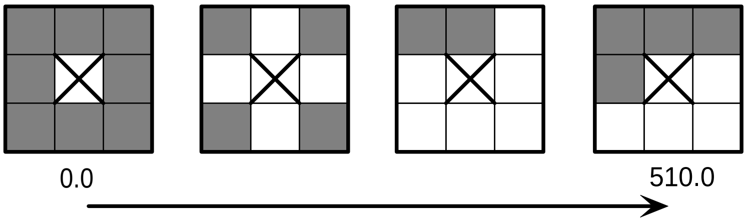

FIGURE 5.5. Several different neighbourhoods around the central pixel. $e(i,j)$ increases from left to right.

5.4.2 Image enhancement

Image enhancement covers several different tasks such as removing or smoothing out noise while preserving or enhancing edges. In the past, the most common approach was to design a filter as sophisticated as possible which performed the enhancement on the complete image.

In recent years, another approach was investigated [CK95]. Instead of using one filter for the whole image, a filter bank with several filters of different characteristics is created. Then, a mechanism has to be developed which decides for each pixel which filter to apply, or, more generally, how to weigh each filter in the filter bank [CK95]. Fuzzy reasoning turned out to be a suitable filter-selecting mechanism; hence, we will follow that approach in this presentation.

5.4.3 Implementation of the enhancement view

The previous presentation of an image segmentation algorithm and its use to enhance images is independent from and not limited to our data/view model. Sure enough, however, I'd like to present this example as a final demonstration of the capabilities of the data/view model.

The first step in order to do this is to create or load an image and to "wrap it around" its boundaries; for this, I used a window view due to its internal usage of a cycle iterator.

In the next step, we have to form the neighbourhood of pixels around the central current pixel. Since we do not only want to calculate $e(i,j)$ out of this neighbourhood, but also apply a filter onto it, the complete 3×3 area is passed to the neighbour view, as shown in LISTING 5.11.

```
typedef boost::view::neighbour_view<WrappedImage, 9> View3x3;
View3x3::difference_type mask3x3( -stride-1, -stride , -stride+1,
                                         -1,        0,         +1,
                                  +stride-1, +stride , +stride+1 );
View3x3 view3x3( wrappedImage, mask3x3 );
```

LISTING 5.11. The "image enhancement view" operates on a 3×3 neighbourhood.

The implementation of the function computing the discrepancy norm of the neighbourhood is relatively straight-forward. The only point to watch is that, although its input values are all integer values in the range $\{0, \ldots, 255\}$, the computation has to be done with floating point precision to avoid rounding errors.

The calculated discrepancy norm and the variation are then taken as input for another function which determines the segmentation type of the area. This function wraps a very simple fuzzy system as described in [BBK96] with two input variables, one output variable, and five rules to decide between one of the four different types.

The final step is to use the computed segmentation type in order to decide which filter to take. The function object which combines all these steps is shown in Listing 5.12.

```
struct ImageEnhancer:
  public std::unary_function< boost::tupple::n_fold_tuple<uchar,9>::type,
                              uchar >
{
  result_type operator()( const argument_type& u ) const
  {
    // Reorder neighbourhood elements as shown in Figure 5.4.
    boost::tupple::n_fold_tuple<uchar,8>::type arg(
      u.get3(), u.get0(), u.get1(), u.get2(),
      u.get5(), u.get8(), u.get7(), u.get6() );

    double dnorm = discrepancyNorm( arg );
    double var   = variance( arg );

    AreaType areaType = determineAreaType( dnorm, var );

    switch( areaType )
    {
    case Homogeneous: return filterPixel( u, ident );
      break;
    case Edge:        return filterPixel( u, sharpen );
      break;
    case Halftone:    return filterPixel( u, blur );
      break;
    case Picture:     return filterPixel( u, smooth );
      break;
    }
  }
};
```

Listing 5.12. First the type of the neighbourhood is determined; then, an enhancing filter is selected accordingly.

What is left is to use this function object. Another view, namely a transform view, does take our ImageEnhancer function and applies it to the wrapped neighbour view. In code, this needs just another two lines:

```
typedef boost::view::transform_view<View3x3, ImageEnhancer> EnhancedView;
EnhancedView enhancedView( view3x3 );
```

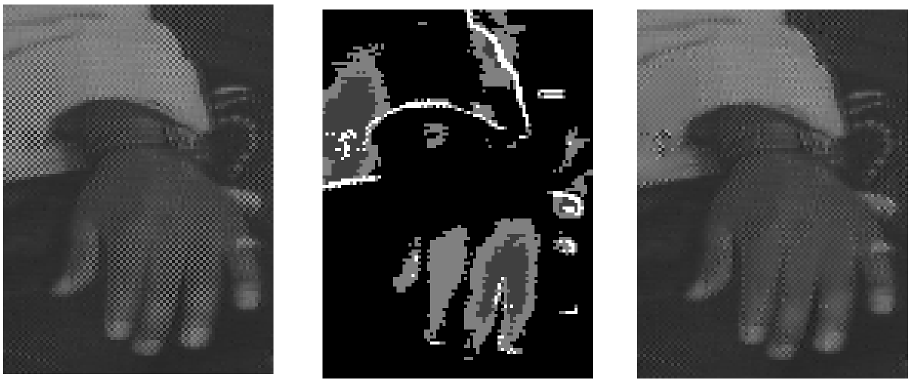

FIGURE 5.6. Original image, segmentation, and enhanced image.

The computations are relatively fast and take about 0.5 seconds for an 768 × 576 grey-level image. FIGURE 5.6 shows one example of a “noisy” image, its segmentation, and the computed enhanced image where the unwanted “chequerboard” artefacts are removed.

BIBLIOGRAPHY

[Ara00] Kaoru Arakawa, *Fuzzy rule-based edge detection using multiscale edge images*, IEICE Trans. Fundamentals **E83-A** (2000), 291–300.

[AS01] David Abrahams and Jeremy Siek, *Policy adaptors and the Boost iterator adaptor library*, Second Workshop on C++ Template Programming, Tampa Bay, Florida, USA, October 14 2001.

[BB00] Christopher Baus and Thomas Becker, *Custom iterators for the STL*, First Workshop on C++ Template Programming, Erfurt, Germany, October 10 2000.

[BBK96] P. Bauer, U. Bodenhofer, and E. P. Klement, *A fuzzy algorithm for pixel classification based on the discrepancy norm*, Proc. FUZZ-IEEE'96, vol. III, 1996, pp. 2007–2012.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker, *Generative programming*, Addison-Wesley, 2000.

[CK95] YoungSik Choi and Raghu Krishnapuram, *Image enhancement base on fuzzy logic*, Proceedings of the 1995 International Conference on Image Processing (ICIP '95), vol. 1, October 1995, pp. 167–170.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley, 1995.

[Jär99a] Jaako Järvi, *ML-style tuple assignment in standard C++ – extending the multiple return value formalism*, Tech. Report 267, Turku Centre for Computer Science, April 1999.

[Jär99b] ———, *Tuples and multiple return values in C++*, Tech. Report 249, Turku Centre for Computer Science, March 1999.

[Jos99] Nicolai M. Josuttis, *The C++ standard library: a tutorial and reference*, Addison-Wesley, 1999.

[Köt] Ullrich Köthe, *VIGRA - Vision with Generic Algorithms*, Cognitive Systems Group, University of Hamburg, Germany.

[Köt99] ———, *Reusable software in computer vision*, Handbook on Computer Vision and Applications (B. Jähne, H. Haußecker, and P. Geißler, eds.), vol. 3, Acadamic Press, 1999.

[Köt00] ———, *STL-Style Generic Programming with Images*, C++ Report Magazine **12** (2000), no. 1.

[KR88] Brian W. Kernighan and Denis M. Ritchie, *The C programming language*, second ed., Prentice Hall, 1988.

[Küh99] Thomas Kühne, *A functional pattern system for object-oriented design*, Forschungsergebnisse der Informatik, no. 47, Verlag Dr. Kovač, 1999, also appeared as: PhD thesis, TU Darmstadt, 1998.

[Mye95] Nathan C. Myers, *Traits: a new and useful template technique*, C++ Report (1995).

[NW87] H. Neunzert and B. Wetton, *Pattern recognition using measure space metrics*, Tech. Report 28, Universität Kaiserslautern, Fachbereich Mathematik, November 1987.

[PW99] Gary Powell and Martin Weiser, *Container adaptors*, Tech. Report SC 99-41, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1999, also appeared in: C/C++ Users Journal, 18 (2000), No. 4,40-51.

[PW00] ———, *VTL (View Template Library) documentation*, http://www.zib.de/weiser/vtl/index.html, 2000.

[Sey95] Jon Seymour, *Views - a C++ Standard Template Library extension*, http://www.zeta.org.au/~jon/STL/views/doc/views.html , 1995.

[SM01] Yannis Smaragdakis and Brian McNamara, *FC++: Functional tools for object-oriented tasks*, Tech. report, College of Computing, Georgia Institute of Technology, 2001.

[Smi97] Steven W. Smith, *The scientist & engineer's guide to digital signal processing*, 1st ed., California Technical Pub., 1997.

[SS00] Jörg Striegnitz and Stephen A. Smith, *An expression template aware lambda function*, First Workshop on C++ Template Programming, Erfurt, Germany, October 10 2000.

[STL] *Standard Template Library Programmer's Guide*, http://www.sgi.com/tech/stl/.

[Str97] Bjarne Stroustrup, *The C++ Programming Language*, 3rd ed., Addison-Wesley, 1997.

[Wey16] H. Weyl, *Über die Gleichverteilung von Zahlen mod. Eins*, Math. Ann. **77** (1916), 313–352, in German.

[WP00] Martin Weiser and Gary Powell, *The View Template Library*, First Workshop on C++ Template Programming, Erfurt, Germany, October 10 2000.

Curriculum Vitae

Roland Richter

Personal information

Name	Roland Richter
Address	Steingasse 18a, 4020 Linz, Austria
Phone	+43-732-77 63 18
Email	roland@flll.jku.at
Homepage	www.flll.jku.at/staff/private/roland/
Birth	23th March 1974, Linz
Citizenship	Austria

Education

1980-84 Primary school in Lichtenberg

1984-92 Grammar school (BRG Fadingerstraße) with emphasis on natural sciences. Final exam with distinction.

since 1992 Studying mathematics at Johannes Kepler University Linz.

Work experience

1997-98 Military service

July 1998- Research assistant at the Fuzzy Logic Laboratorium Linz (FLLL), working in the area of signal and image processing within a long-term partnership between the FLLL, Sony DADC Austria, and Uni Software Plus (USP).

March 2000- Project manager of project “Inspire” , a joint effort of FLLL, Software Competence Center Hagenberg (SCCH), and USP, with the goal of developing a real-time print inspection and quality assurance framework.