ArrayOps
C++ Vector Computation
Manual

Magnus Erik Hvass Pedersen

May 19, 2006

ArrayOps C++ Vector Computation Source-Code Library.
The Manual, Second Edition by Magnus Erik Hvass Pedersen.

Contents

Preface

My first need for vector computation in the C++ programming language, arose years ago when I was working for a company making software-based audio synthesizers. Since then, I have used vector computations in other contexts also, but many of the semantic requirements for ArrayOps, were uncovered back then.

I am a strong believer in keeping things simple. In fact, I believe that if something gets too complex, it is often because of lack of insight. I have tried to keep the ArrayOps implementation as simple as currently possible (with the limitations of C++ in mind), so as to easen maintenance of the source-code library, and thus decrease the risk of bugs.

I also believe in cooperation, so should you find any bugs or have any suggestions for improvements, you are strongly encouraged to share your findings, and participate in the further development of ArrayOps — this also goes for the manual you are now reading. Godspeed!

Magnus Pedersen, Copenhagen, March 2006

Chapter 1

Introduction

1.1 Overview

This manual describes the usage and implementation of the ArrayOps library for performing vector computations in the C++ programming language. The manual is divided into the following chapters:

- Chapter 1 first discusses the need for having such a library, and then describes libraries that are similar to ArrayOps, giving reasons for when and why ArrayOps may be useful. The key concepts of ArrayOps are outlined, along with the syntactic and semantic principles of the library. This chapter also has information about the license under which the library and this manual are published, as well as information on where to find updates to the library and manual, and how to contact the developers of the library.
- Chapter 2 is a reference manual as well as a brief tutorial on the library's intended usage. This chapter should be sufficient for most users of the ArrayOps library.
- Chapter 3 describes how the ArrayOps library is implemented, and how to modify and extend the library. This chapter is probably only of interest to more experienced users, and requires a good understanding of template programming in the C++ language.

1.2 Motivation

The inventor of the C++ programming language, Bjarne Stroustrup, clearly expressed the need for supporting vector computations in C++, as follows [Stroustrup, 1991, Section 22.4]:

> Much numeric work relies on relatively simple single-dimensional vectors of floating-point values. In particular, such vectors are well

> supported by high-performance machine architectures, libraries relying on such vectors are in wide use, and very aggressive optimization of code using such vectors is considered essential in many fields.

However, it was not until recently, that C++ had matured sufficiently to facilitate efficient implementation of such numeric vectors. The original vector-support in C++ (namely the socalled `std::valarray`-class), was grossly inefficient, and in some cases even unusable, for example when implicit allocations were not allowed.

Naturally, Stroustrup's intention was for the `std::valarray`-class to be specialized for each machine architecture by the compiler implementors; but this often did not happen.

1.3 Modern Implementations

With the advent of socalled meta-programming, it is possible for the compiler to generate specialized code at compile-time. This is utilized by a number of source-code libraries for a number of different things, including vector computations.

1.3.1 Existing Libraries

Many software developers will try and get you to use their libraries exclusively. Naturally there is personal gratification in having other people benefit from your hard work, but this kind of monopoly, runs the risk of stagnating the development, of the very library that one is trying to promulgate.

To stress the fact that ArrayOps is not considered to be the *be-all and end-all* of vector computation libraries, you are strongly encouraged to consider whether some of the other vector computation libraries are more suitable for your needs:[1]

- Blitz++ [Veldhuizen] is a commonly known library, and its author is reported as having been one of the pioneers in meta-programming for the C++ programming language. Blitz++ supports multi-dimensional vectors (which ArrayOps currently does not), and reportedly implements different optimizations in its framework, including loop-unrolling. The Blitz++ implementation appears vast and complicated, and moreover, is not very well documented. Blitz++ has two datatypes, one for vectors of arbitrary sizes, and one for vectors of smaller sizes which must be known at compile-time. These may *not* be combined in a single arithmetic expression. Blitz++ also does a lot of checking and implicit resizing at runtime.

- POOMA [Karmesin et al.] is similar to Blitz++, and also supports multi-dimensional vectors. Its reference manual however, is much more thorough

[1] Please note that I am not an expert in any of these libraries, and this overview may not be entirely correct. Feel free to inform me of any errors, or if you know of any other libraries that should be added to this list.

than that of Blitz++. POOMA also supports distributed computation, currently implemented using the socalled *Message Passing Interface* (MPI) [committee]. The POOMA implementation is also very large and complicated, and even appears to be using a proprietary programming language of its own, to generate the many source-code files.

- SVMTL [Tisdale] is the *Scalar, Vector, Matrix and Tensor* class-library, which does not appear to be using meta-programming at all. In fact, it does not even appear to be using templates. In this regard, it is very different from Blitz++, POOMA, and ArrayOps, and quite possibly suffers from some of the same drawbacks as the `std::valarray` class. The SVMTL library provides a multitude of types though, for instantiating various kinds of mathematical objects (scalars, vectors, etc.), and with elements of different datatypes (integers, floating points, and so on). Because SVMTL does not use templates and meta-programming, it may be supported by a greater number of C++ compilers.

The reason why Blitz++ and POOMA are so complicated in their implementations when compared to ArrayOps, is probably that they do not use socalled *Reverse Inheritance* (see section 1.4.5 or chapter 3), which greatly increases the flexibility of the framework, and reduces the amount of nearly redundant code.

1.4 The ArrayOps Library

There are a number of justifications for making ArrayOps. For example in regards to the implementations themselves, the other libraries are immensely large and difficult to comprehend – probably also for their original developers. This increases the maintenance difficulties, and hence the risk of bugs, and also makes it practically impossible for someone else to correct and extend the libraries.

In terms of practical usability, the other libraries also suffer in a number of ways. Just take an obviously desirable feature, such as combining different kinds of arrays in a single arithmetic expression, that is, combining arrays which use different kinds of storage-mechanisms; which is supported directly by the ArrayOps framework, and therefore works for your own array-implementations as well, without you having to change the framework.

Also when it comes to the legal issues, you may find the other libraries lacking. Usually they are published under the GNU General Public License, which does not allow for you to link the source-code library into commercial programs, and distribute those programs at a profit. ArrayOps was specifically developed with this in mind, both in terms of features and the legal license.

The only real drawback of ArrayOps compared to those other libraries, is that it currently does not support multi-dimensional arrays.

1.4.1 Terminology & Notation

For historic reasons, the `valarray`-class in C++, was called an array instead of a vector (recall that a vector in C++, is a resizable datastructure for holding elements of arbitrary type, whereas numeric vectors are often only practical for holding numbers). This practice was adobted by similar numeric libraries, and to keep things simple, ArrayOps uses the same terminology. So from here on out, we shall use the word array instead of vector, even when speaking in mathematical terms.

In regards to the source-code notation, a slightly different and more modern approach has been chosen however, in that the names of classes and functions always begin with a capital letter – unless there is some specific reason not to do this. This notational style is believed to be more readable than the usual old-school C++ style.

Also, you will often be able to read from the name of a class or a function, how many arguments it takes. For example, the class for a unary expression is named `Expr1`, and the class for a binary expression is named `Expr2`.

1.4.2 Principles

In the development of ArrayOps, it was the intention to keep the syntax and semantics as close as possible to C++, while retaining a framework that was simple and could (somewhat) easily be maintained and extended. Other than that, the guiding principles in the design of ArrayOps, were as follows:

- No implicit allocations can take place, unless the user (that is, the application programmer) explicitly allows for this to happen, by using a specialized array-class for that purpose.
- No implicit initialization is performed, other than that performed by C++ itself (which is usually none, depending on the compiler and whether you are compiling the program in debug- or release-mode).
- No side-effects are allowed in arithmetic expressions. This imposes some unfortunate restrictions, but encourages a style of programming that is easier to understand and maintain, and usually also implies fewer bugs.
- Array-sizes are only checked in debug-mode. When this checking is not enough, you may wrap the ArrayOps code in exception-handling statements by yourself, at a slight cost of execution speed.
- ArrayOps is strong-typed to eliminate unintentional degradations in numeric precision.

In general, whenever there was a choice of making the semantics of ArrayOps either loose or rigid, rigidity was chosen. Though often there was no choice, due to the specific way the framework is implemented. This rigidity is believed to strengthen the user's programming style, thus causing fewer bugs – and hopefully you will learn to live with the shortcomings.

1.4.3 Flattened Loops

The primary idea with ArrayOps, is to have the compiler automatically transform arithmetic expressions involving numeric arrays, into flattened loops. Take for example the expression:

```
A = r * B + s * C;
```

where `A`, `B`, and `C` are arrays of, say, integers, and `r` and `s` are some scalar integer variables. To circumvent all the problems with `std::valarray`, we would ultimately like the compiler to see this expression as a single flattened loop, instead of `valarray`'s sequence of loops that compute intermediate values, and hence requires temporary storage as well. That is, we would like for the expression `A = r * B + s * C;` to compile into the following loop:

```
for (unsigned int i=0; i<A.Size(); i++)
{
    A[i] = r * B[i] + s * C[i];
}
```

To do this, we must employ a number of advanced object-oriented programming techniques, which will be briefly introduced next, and described more thoroughly in chapter 3.

1.4.4 Template Meta-Programming

Recall that C++ supports template-classes and -functions, where some or all of the datatypes may be provided as socalled template arguments. These template arguments must be provided at compile-time, so the compiler can specialize the source-code for that particular datatype. Furthermore, the C++ compiler is sometimes able to deduce the template arguments for template functions, according to the arguments passed to that function.

In template meta-programming we use this to have the compiler build specialized datastructures and functions for us, by combining simpler building-blocks that we have provided, as well as directions on how to combine them. This means the compiler is able to create specialized code that is inlined, and should thus be more efficient than, say, virtual functions. The particulars of the meta-programming framework in ArrayOps, are covered in chapter 3.

1.4.5 Reverse Inheritance

Another little trick that is employed by ArrayOps, and which allows us to implement much less than would ordinarily be required, is socalled *reverse inheritance*. Once again we use template arguments to have the compiler specialize the source-code during compilation, and the idea is basically to provide a class as a template-argument to another class, and have the latter inherit from that template-argument. It may sound a bit confusing, but is actually fairly simple,

although it naturally makes the source-code a bit harder to understand, than if we had used regular C++ inheritance.

The benefit of using reverse inheritance, is that we can make specializations of classes with inlined functions, and thus increase performance. If we were to use virtual functions – which is the usual way of specialization in C++ – we would experience a severe penalty in the performance of ArrayOps. So reverse inheritance enables us to re-use the entire framework of ArrayOps for different array-types, without any loss in performance; which again translates into ArrayOps being much easier to maintain and extend.

1.4.6 Macros

Macros are not commonly used in C++, as they are very weak language constructs, that give rise to bugs that may be hard to locate. However, when used sparingly inside a well-tested framework, they sometimes provide great reductions in the amount of (more or less) redundant source-code. This is exactly what ArrayOps uses macros for, but the user is currently discouraged from using these internal ArrayOps macros, as they are likely to change in the future.

1.4.7 Optimizations

ArrayOps does not provide any low-level optimizations, other than the flattening of loops as described in section 1.4.3. That is, ArrayOps does not perform socalled loop-unrolling, or anything like that.

The reason is that most modern C++ compilers do this already, and it would be grossly redundant to support the very same features in ArrayOps, that you may control via compiler flags and arguments. This would only have meant a more complicated framework, and maybe even be counter-productive, in the sense that the C++ compiler would have trouble figuring out what is going on, and where it could apply its own optimizations.

In other words, optimization of the ArrayOps code, is left entirely up to the C++ compiler. This also includes the use of socalled SIMD instructions,[2] which are supported by some platforms and not others. It would be possible to implement support for SIMD instructions directly in ArrayOps, but it gets increasingly difficult with all the index mapping features and so on (see section 2.4).

SIMD optimizations are also beginning to appear as options in C++ compilers, and it is therefore suggested, that you instead of making a single program that automatically switches between SIMD and non-SIMD implementations, simply compile a version of the program for each platform you wish to support, then turn on all the platform-specific optimizations during each compilation, and finally have your installation-program choose the correct version of the program to install on a given machine.

[2] SIMD stands for Single-Instruction-Multiple-Data, and is a vector-processing computional unit, that may perform the same operation on several pieces of data in a single go.

1.4.8 Testing

In my years of software development, I have many times experienced that joyous but treacherous feeling over something finally being finished and working, just to find a bug two seconds later. My attitude towards testing has therefore become, that the software is only bug-free, until another bug is discovered. This should not be taken as a sign of poor quality or pessimism, but rather as a sign of long and bitter experience. I would think most experienced software developers have gone through the same motions I have.

So to convince yourself that ArrayOps is working for the purpose that you intend to use it for, I suggest you conduct two phases of testing. First comes the testing of the basic ArrayOps features, to establish whether the library appears to be working at all. I have provided a small test-program for this, and you are most welcome to share any alterations you might have. Running the test-program in debug-mode, the output should be manually scrutinized by yourself, step by step. Make sure to save the output, which you should also compare to the test-program's output when compiled and run in release-mode.

After having convinced yourself that ArrayOps seems to be fairly soundly implemented, you should test it in your own application. Once again, I suggest that you first test your program in debug-mode (which enables a lot of testing in the ArrayOps source-code itself), and after that, test the program in release-mode. After a sufficient amount of testing, you may conclude that your program appears to be working as expected. But remember that the objective of testing is not to prove that your program is working or is bug-free; the objective of testing, is to uncover bugs! This attitude is important, because otherwise you might subconsciously only run tests which you feel are safe and that the program can handle.

1.5 License

ArrayOps is intended to be used (and improved) by as many people as possible, and both in scientific and professional settings, without imposing any responsibility on the author(s) of ArrayOps. Should you require an exception to either the source-code or manual licenses described below, please contact the author(s) for obtaining more specialized licenses.

1.5.1 Source-Code License

The ArrayOps source-code is published under the *GNU Lesser General Public License* [Foundation], which essentially means that you may distribute commercial programs that link with the ArrayOps library, as well as make alterations to the ArrayOps library itself. There are certain terms to be met though, but please see the license for those details. Note that you should use the license included in the ArrayOps source-code distribution.

1.5.2 Manual License

I have been unable to find a standard license for protecting the rights to this manual in a satisfactory manner. Generally speaking, you may download, print, and use the manual for any personal purpose, be it commercial or non-commericial – provided the author(s) are not held responsible for your actions, or any damage caused by your use of the manual.

If you want to distribute the manual commercially, for example in a printed book, or on a web-page that requires payment, then you must obtain a license from the author(s). If you wish to make alterations or additions to the manual, you may contact the author for a copy of the LaTeXsource-files, and coordinate your effort with the other author(s).

1.6 Contact

To obtain updates to the ArrayOps source-code library, or to get newer revisions of this manual, you should check the library's webpage (`http://www.Hvass-Labs.org/`). There you may also find information on how to contact the authors, where to ask technical questions, obtain individual licenses for specific purposes which are not covered by the generic license, and so on.

Chapter 2

Reference Manual

2.1 Introduction

This chapter describes the facilities of the ArrayOps library, and gives examples on how to use them.

2.2 Installation

To use the ArrayOps source-code library, you must first add the proper path to your C++ project's include-paths. This is described in more detail in the installation instructions that come with the library, and is therefore omitted here.

2.2.1 Including Just Header-Files

Note however, that all files in the ArrayOps source-code library are actually header-files, which means you should just include the header-file for the array-type that you want to use, and that header-file will automatically include the required framework, the mathematical operator and function overrides, and so on. That is, there is no code-library that requires for you to build it, so installing and using ArrayOps is made as simple as possible.

2.3 Array Types

One clear advantage of ArrayOps, is that you may combine different kinds of arrays in a single arithmetic expression. This version of ArrayOps has the following array-types:

- `Array` is the basic, resizable array. See section 2.3.2 for details.
- `ArrayMini` is an array whose size is small, and known at compile-time. See section 2.3.3 for details.

- `ArrayUse` takes a user-supplied storage and uses it for its array-elements. See section 2.3.4 for details.
- `ArrayAuto` is the same as the basic `Array`-class, only it automatically resizes according to the size of the right-hand expression in an assignment. See section 2.3.5 for details.

Accessing an array-element takes constant time in all array-types of the ArrayOps library, unless otherwise noted.

2.3.1 ArrayBase

By definition, all array-types which may occur as left-hand values (also called lvalues) in assignments, must derive from the `ArrayBase`-class. As a user of the ArrayOps library, you would normally not worry about this class, but it is useful for you to know of its existence, when you need to understand what you can and can not do with arrays and arithmetic expressions.

Most importantly is perhaps that an instance of the `ArrayBase`-class must be sized. That is, it must provide a function `Size()` which returns some appropriate value, so that you may address elements indexed between zero and `Size()` minus one. This may not seem all that important right now, but is an important assumption in many regards.

2.3.2 Array

The `Array`-class is the basic resizable array, and is found in the `<ArrayOps/Array.h>` header-file, and takes the following template-arguments:

- `T`, which is the datatype of the array's elements (e.g. `int` or `double`).
- `Parallel`, which is a boolean designating whether assignment-operations should be performed in parallel, whenever this array-instance is the left-hand of an assignment. `Parallel` defaults to `true` for the `Array`-class.

Constructors

There are two constructors available for the `Array`-class:

- `Array()`, which takes no arguments and performs no allocation. You must call `Resize()` before performing any operations on the array.
- `Array(unsigned int size)`, which takes as argument the size of the array, and allocates storage accordingly. The size can be zero.

The destructor deletes the allocated storage (if any) upon deletion of the `Array`-object. Also note that in debug-mode, an assertion will ensure that no operations are performed on an array whose storage has not yet been allocated.

Additional Functions

Furthermore, the `Array`-class provides the following functions:

- `Resize(unsigned int size)`, that takes as argument the new size of the array (which can be zero), and first deletes the current storage (if any), and then tries to allocate the new storage. If this fails, an exception is raised, and the old storage is lost. If allocation succeeds, the new storage is not initialized though, as you must do this explicitly.
- `ResizeCopy(unsigned int size)`, that takes as argument the new size of the array (which can be zero), and first tries to allocate new storage. If this fails, an exception is raised, but the old storage is retained and not lost. If allocation succeeds, data is copied from the old storage, and any remaining elements are left uninitialized, as you must do this explicitly. Then the old storage is deleted.

2.3.3 ArrayMini

The `ArrayMini`-class is for arrays whose sizes are known at compile-time. The arrays are typically allocated on the execution-stack, which means the arrays should be fairly small. The class is found in the `<ArrayOps/ArrayMini.h>` header-file, and takes the following template-arguments:

- `T`, which is the datatype of the array's elements (e.g. `int` or `double`).
- `kSize`, which is an unsigned integer designating the size of the array, and must be known at compile-time. The size must be one or greater.
- `Parallel`, which is a boolean designating whether assignment-operations should be performed in parallel, whenever this array-instance is the left-hand of an assignment. `Parallel` defaults to `false` for the `ArrayMini`-class.

Constructors

There is just a single constructor available for the `ArrayMini`-class:

- `ArrayMini()`, which takes no arguments.

2.3.4 ArrayUse

The `ArrayUse`-class is used when you are supplied with a C++ array from somewhere (e.g. the operating system), and wish to use it within the ArrayOps framework, but without having to copy it first. The class is found in the `<ArrayOps/ArrayUse.h>` header-file, and takes the following template-arguments:

- `T`, which is the datatype of the array's elements (e.g. `int` or `double`).

- `Parallel`, which is a boolean designating whether assignment-operations should be performed in parallel, whenever this array-instance is the left-hand of an assignment. `Parallel` defaults to `true` for the `ArrayUse`-class.

Constructors

There is just a single constructor available for the `ArrayUse`-class:

- `ArrayUse (T* storage, unsigned int size)`, which takes as arguments a pointer to the storage, and its size (i.e. the number of elements). Note that there is no way for the `ArrayUse`-class to check whether the storage actually has that size, so the developer must ensure this by herself.

Also note that the storage is not copied, the `ArrayUse`-instance merely uses the storage that is supplied in its constructor.

2.3.5 ArrayAuto

The `ArrayAuto`-class derives from the `Array`-class from section 2.3.2, and provides the exact same functions and has the same template arguments. It is found in the `<ArrayOps/ArrayAuto.h>` header-file.

Automatic Resizing

As mentioned above, the purpose of the `ArrayAuto`-class is to automatically resize itself according to the size of the right-hand expression in an assignment. This is done by using the functions `Resize` and `ResizeCopy` from the `Array`-class above, depending on whether an overwriting or accumulative assignment is performed.

Resizing in Accumulative Assignments

When the right-hand expression of an accumulative assignment, is of a greater size than the `ArrayAuto`-object occuring on the left-hand, and since the `ResizeCopy`-function does not initialize any newly allocated elements of the array, the remaining part of the resized array will contain garbage, after performing such an accumulative assignment.

This may change in future versions of ArrayOps, but you should generally be careful when using `ArrayAuto` in accumulative assignments – and whenever possible you should use the `Array`-class instead, and manually resize the array when needed, so as to improve both safety and efficiency.

Constructors

There are several constructors available for the `ArrayAuto`-class, as we also allow for implicit initialization from other `ArrayAuto`-objects as well as more general ArrayOps expressions:

- `Array()`, which takes no arguments and performs no allocation. You must call `Resize()` before performing any operations on the array.
- `Array(unsigned int size)`, which takes as argument the size of the array, and allocates storage accordingly. The size can be zero.
- `ArrayAuto(ArrayAuto const& x)`, which takes another `ArrayAuto`-object as argument, and allocates storage accordingly, and then copies the elements from that object.
- `template <class S1>`
  `ArrayAuto(Expr<T, S1> const& x)`, which takes an arbitrary ArrayOps expression as argument, allocates storage accordingly, and then evaluates the supplied expression, assigning the result to the newly constructed `ArrayAuto` object.

The destructor deletes the allocated storage (if any) upon deletion of the `ArrayAuto`-object. Also note that in debug-mode, an assertion will ensure that no operations are performed on an array whose storage has not yet been allocated.

2.3.6 Accessing Array Elements

Array elements are normally accessed as you would with any basic C++ array. That is, if `A` is an ArrayOps array, we would write `A[i]` to address the *i*'th element of that array. Arrays in the ArrayOps library are indexed from zero up to the array's size minus one (also as in C++).

Using standard C++ syntax has a number of advantages, including that you do not need to rewrite source-code written for C++ arrays (except for maybe the function declaration).

Example

Now, assuming ArrayOps is in your include-path, using it is fairly simple. First let us see how to instantiate an array of `double`-typed elements, consisting of 1024 elements, and assigning values to those elements:

```
#include <ArrayOps/Array.h>          // Include the header-file.

int main(int argc, char* argv[])
{
    const unsigned int k = 1024;     // Size of the array.
    ArrayOps::Array<double> A(k);    // Create the array.

    for (unsigned int i=0; i<k; i++) // Assign values to the
    {                                // array's elements, one
        A[i] = 1.0/(i+1);            // at a time.
    }
```

```
    return 0;                          // Return from main.
}
```

2.3.7 Checked Access

Array elements may also be accessed using the `at()`-function, which first checks if the supplied index is within bounds, and if not, raises an exception. The index must (as usual) be between zero and the array's size minus one. Both `const` and non-`const` `at()`-functions are provided, but the non-`const` lookup may currently only be executed on instances of the `ArrayBase`-class, whereas the `const` lookup may be executed on all ArrayOps expressions:

- `T& at(unsigned int i)`, which is the non-`const`-version.
- `T const& at(unsigned int i) const`, which is the `const`-version.

If the index is out of bounds, the exception `std::out_of_range` will be raised with an appropriate error-message.

Example

The following example shows how the `at()`-function may be used with a `try catch` clause. Recall that arrays are indexed from zero to their size minus one, which means the valid indices for the array `A` range from zero up to `k-1`, so trying to access the element with index `k` is clearly a mistake:

```
const unsigned int k = 1024;     // Size of the array.
ArrayOps::Array<int> A(k);       // Create the array.

try
{
    A.at(k) = 12345;             // Throws exception!
}
catch (...)
{
    std::cout << "Oops" << std::endl;
}
```

And the following example shows how to use checked lookup on an arithmetic expression:

```
const unsigned int k = 1024;     // Size of arrays.
ArrayOps::Array<int> A(k), B(k); // Create arrays.

// ...

try
```

```
{
    std::cout << (A+B).at(0) << std::endl;
}
catch (...)
{
    std::cout << "Oops" << std::endl;
}
```

This way of accessing individual elements in an arithmetic expression involving ArrayOps arrays, may also be done with ordinary and non-checked `operator[]` lookup:

```
std::cout << (A+B)[0] << std::endl;
```

Both of which of course return a `const`-references, since an expression such as `A+B` is not a socalled lvalue, and it therefore does not make any sense to assign a value to it.

2.4 Index Manipulators

An array may also be accessed through the following index-manipulators:

- `Slice` takes as arguments an index offset and a new array size, and creates a slice of the array accordingly.
- `Cycle` takes as argument an offset, and creates a cyclic array of the same size as the original array, and with the given index offset.
- `Reverse` takes no arguments, but merely reverses the order of the array's elements.

Note that none of these index-manipulators copy any elements from the arrays, but merely provide a wrapper-object that maps the index for accessing the underlying array-elements.

As a result of this, you should generally avoid using the same array in the left- and right-hand expressions of an assignment, when you also apply index manipulators, as the result may be a bit surprising or even unpredictable. The reason is that ArrayOps does not assume any particular traversal order when computing an expression involving arrays; and this is particularly true if you use parallelism as well.

2.4.1 Slice

A slice of an array is itself an array, which is (possibly) of smaller size and whose index is (possibly) offset from that of the original array.

Instantiating

The `ArraySlice`-class resides inside the `ArrayBase`-class, whilst also inheriting from `ArrayBase` itself (this is possible through the use of forward-declarations). The class has no template-parameters beyond those of the `ArrayBase`-class. It is most easily instantiated through the `Slice`-function from `ArrayBase`:

- `Slice(unsigned int offset, unsigned int size)`, which takes as arguments the offset and the size of the new array. It returns an `ArraySlice`-object, which may be used directly, or stored for later re-use (see example below).

Note once more, that an `ArraySlice` is merely an index-mapping, and does not copy the elements of the original array.

Example

The following example instantiates an array `A`, creates a slice half its size, and offset a quarter of `A`'s size, and initializes the elements of that slice, to a certain value:

```
const unsigned int k = 1024;                    // Array size.
Array<int> A(k);                                // Actual array.
Array<int>::ArraySlice B = A.Slice(256, 512);  // The slice.

for (unsigned int i=0; i<B.Size(); i++)         // Initialize
{                                               // slice with
    B[i] = i;                                   // some values.
}
```

Note that the elements of the slice `B` are really elements in the array `A`. That is, the value of `B[0]` is really `A[256]` which is set to 0 in the loop above, `B[1]` is really `A[257]` which is 1, `B[2]` is really `A[258]` which is 2, and so on; because the elements of slice `B` merely refer to the elements of the original array `A`.

2.4.2 Cycle

A cycle of an array is itself an array, whose index is offset from that of the original, but which allows the index to grow indefinitely, as it is mapped back to the proper range by using modulo-arithmetics. The size of a cyclic array is the same as the original array.

Instantiating

The `ArrayCycle`-class resides inside the `ArrayBase`-class, whilst also inheriting from `ArrayBase` (this is possible through the use of forward-declarations). The class has no template-parameters beyond those of the `ArrayBase`-class. It is most easily instantiated through the `Cycle`-function from `ArrayBase`:

- `Cycle(unsigned int offset)`, which takes as argument the offset from which to start the cycle. It returns an `ArrayCycle`-object, which may be used directly, or stored for later re-use (see example below).

Note that an `ArrayCycle` is merely an index-mapping, and does not copy the elements of the original array.

Example

Similarly to the example for the `Slice`-function above, we may create a cycle offset to, say, index 1022:

```
const unsigned int k = 1024;                    // Array size.
Array<int> A(k);                                // Actual array.
Array<int>::ArrayCycle C = A.Cycle(1022);       // The cycle.

for (unsigned int i=0; i<C.Size(); i++)         // Initialize
{                                               // slice with
    C[i] = i;                                   // some values.
}
```

The elements of the cycle `C` are really elements in the array `A`. That is, the value of `C[0]` is really `A[1022]` which is set to 0 in the loop above, `C[1]` is really `A[1023]` which is 1, `C[2]` is really `A[0]` which is 2, and so on.

2.4.3 Reverse

The reverse of an array is itself an array, whose indices are merely inverted to go from `Size()-1` down to zero. The size of a reversed array is the same as the original array.

Instantiating

The `ArrayReverse`-class resides inside the `ArrayBase`-class, whilst also inheriting from `ArrayBase` (this is possible through the use of forward-declarations). The class has no template-parameters beyond those of the `ArrayBase`-class. It is most easily instantiated through the `Reverse`-function from `ArrayBase`:

- `Reverse()`, which takes no arguments and returns an `ArrayReverse`-object, which may be used directly, or stored for later re-use (see example below).

Note that an `ArrayReverse` is merely an index-mapping, and does not copy the elements of the original array.

Example

Similarly to the example for the `Slice`-function above, we may create a reversed array:

```
const unsigned int k = 1024;                    // Array size.
Array<int> A(k);                                // Actual array.
Array<int>::ArrayReverse D = A.Reverse();       // Reversed array.

for (unsigned int i=0; i<D.Size(); i++)         // Initialize
{                                               // slice with
    C[i] = i;                                   // some values.
}
```

The elements of the reversed array `D` are really elements in the array `A`. That is, the value of `D[0]` is really `A[1023]` which is set to 0 in the loop above, `D[1]` is really `A[1022]` which is set to 1, `D[2]` is really `A[1021]` which is set to 2, and so on.

2.4.4 Nesting

What if we want to create a reversed slice? That is, first make a slice and then reverse it. We could mistakenly try and write something like the following:

```
Array<int>::ArraySlice::ArrayReverse E =
                                    A.Slice(256, 512).Reverse();
```

But this is erroneous usage of the `Slice` and `Reverse` functions! The reason is that the function call `A.Slice(256, 512)` creates and returns a temporary object, on which the `Reverse` function is invoked. This creates an `ArrayReverse`-object which is then assigned to `E`. After this, the temporary `ArraySlice`-object is automatically destroyed. So `E` now references a temporary object that was just destroyed. The way to write this correctly would therefore be:

```
Array<int>::ArraySlice Z = A.Slice(256, 512);
Array<int>::ArraySlice::ArrayReverse E = Z.Reverse();
```

Where `Reverse` is now invoked on a local object `Z`, which `E` then references. We may of course also use the results of the `Slice` and `Reverse` functions directly in an arithmetic expression, such as:

```
C = B + A.Slice(256, 512).Reverse();
```

assuming `B` and `C` are appropriately typed and sized arrays. Here, the temporary objects created by the calls to `Slice` and `Reverse`, will survive until the entire assignment expression has been evaluated, due to C++ semantics for temporary objects (see page 41 for details).

2.5 Operators

All array-types in ArrayOps may be combined in arithmetic expressions involving various operators – provided of course, that the elements of those arrays support those operators, as the operators are applied on an element-by-element basis.

Note that an essential feature of ArrayOps, is that meta-programming ensures that no actual evaluation of an expression takes place before that expression is assigned to an array. (More on this in section 2.5.4 below.)

2.5.1 Arithmetic Operators

The usual arithmetic operators are provided, with the same meaning as for scalar values: `+`, `-`, `*`, `/`, and `%`. Different array-types may be combined along with constant scalar values and variables, etc.

Example

Let $r, s \in \mathbb{R}$ be two scalar real-values, and $A, B \in \mathbb{R}^k$ be two arrays each of length $k = 16$. Then consider the following expression:

$$r \cdot A + \frac{s}{B}$$

Which means we must compute the expression for each element of the arrays A and B, as follows:

$$r \cdot A_i + \frac{s}{B_i}, \forall i \in \{0, \cdots, k-1\}$$

Now, if we use the double-precision floating point type `double` to approximate real-numbers, and for the sake of illustration, use two different array-types for the arrays A and B, we would have something like the following in C++ using ArrayOps:

```
const unsigned int k = 16;            // Array-size.
Array<double> A(k);                   // Array A.
ArrayMini<double, k> B;               // Array B.
double r, s;                          // Scalar values.

// ...

r * A + s / B;                        // The expression.
```

Note however, that because no assignment occurs, nothing is actually ever computed by the expression `r * A + s / B;` and in section 2.5.4 below, we will see what happens when this is finally assigned to some array.

2.5.2 Bitwise Operators

The usual bitwise operators are provided, with the same meaning as for scalar values: `<<`, `>>`, `&`, |, and `^`. These operators are (usually) only defined for integer-typed values.

2.5.3 Logical Operators

The usual logical operators are provided, with the same meaning as for scalar values: `==`, `!=`, `>`, `<`, `>=`, `<=`, `&&`, ||, and `!`. These operators may take argument-expressions of any (appropriate) datatype, e.g. arrays with elements of type `float`, `double`, or `int` – or any other datatype, as long as it supports the comparison operator in question. Note however, that all these logical operators are assumed to return boolean-typed values.

As usual, these operators work on an element-by-element basis, which means that we would e.g. compute a sequence of comparisons using the operator `==` for arrays `A` and `B`, and not finally *and* the results of these individual comparisons, to create a single measure of how one array compared to another. In other words, the output of applying a logical operator, is itself an array.

Example

Let us say we wish to find out if the elements of some array `A`, are less than the values of some array `B` plus a constant value `c`. This could be done using ArrayOps as follows:

```
const unsigned int k = 16;            // Array-size.
Array<int> A(k), B(k);                // Arrays.
const int c = 12345;                  // Some value.

// ...

A < (B+c);                            // The comparison.
```

Again however, nothing is actually ever computed by the `A < (B+c);` expression, as it contains no assignment operator.

2.5.4 Assignment Operators

The usual assignment operators are provided, with the same meaning as for scalar values: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, and `>>=`. Some of these are only meaningful for integer-type arrays (e.g. the bitwise shifting operators).

Deferred Computation

As previously mentioned, an important aspect of ArrayOps, is that computation of an expression involving array-types, is deferred until assignment of that arithmetic expression to an array.

The advantage of this, is of course that we may iterate through the arrays in the right-hand expression of the assignment operator, and compute the value of the arithmetic expression for each index of the array. This avoids the need for temporary arrays, as is a known problem in the `std::valarray` class of C++.

Nested Assignment

It is possible to nest assignment operators in the C++ programming language. This is also possible in ArrayOps, but there is currently no difference between writing the assignments one by one, or nesting them; they compile to the same thing.

It is possible to change the ArrayOps framework so that a nested assignment is also flattened into a single loop, but it will increase the complexity of the ArrayOps framework considerably, which is considered undesirable.

Example, Assignment Of Arithmetic Expression

Let us modify the examples from before. First, let us say we wish to compute the following mathematical expression, for arrays A, B, and C, and scalar values r and s:

$$C = r \cdot A + \frac{s}{B}$$

This could be implemented using ArrayOps as follows (again assuming the size $k = 16$, and using different array-types):

```
const unsigned int k = 16;            // Array-size.
Array<double> A(k);                   // Array A.
ArrayMini<double, k> B, C;            // Arrays B, C.
double r, s;                          // Scalar values.

// ...

C = r * A + s / B;                   // The expression.
```

The computation of the expression `C = r * A + s / B;` is automatically transformed into a single loop by the ArrayOps framework:

```
for (unsigned int i=0; i<C.Size(); i++)
{
    C[i] = r * A[i] + s / B[i];
}
```

This transformation takes place internally in the compiler, and the textual expression is not actually changed in the source-code.

Example, Assignment Of Comparison Expression

The comparison example from above, can be written as follows, where we assign the result of the comparison to an array `C`:

```
const unsigned int k = 16;            // Array-size.
Array<int> A(k), B(k);                // Arrays A, B.
Array<bool> C(k);                     // Bool array.
const int c = 12345;                  // Some value.

// ...

C = A < (B+c);                        // The comparison.
```

The computation of the expression `C = A < (B+c);` is again transformed into a single loop by the ArrayOps meta-programming framework:

```
for (unsigned int i=0; i<C.Size(); i++)
{
    C[i] = A[i] < (B[i]+c);
}
```

Note that ArrayOps is strong-typed, and as the comparison operator `<` yields a boolean-valued expression, then the array `C` must be boolean-typed also; even though the C++ programming language allows implicit conversions between, say, boolean and integers.

Example, Nested Assignments

Let us say we have the following ArrayOps code-fragment:

```
const unsigned int k = 1024;          // Array-size.
Array<int> A(k), B(k), C(k);          // Arrays.
int m, n;                             // Some values.

// ...

C = m + (A = n * B);                  // The expression.
```

Since the ArrayOps framework treats nested assignments as a series of separate assignments, the expression `C = m + (A = n * B);` is transformed into one loop for each assignment occuring in it. That is, the expression is transformed into the two following loops:

```
for (unsigned int i=0; i<A.Size(); i++)
{
    A[i] = n * B[i];
}

for (unsigned int i=0; i<C.Size(); i++)
{
    C[i] = m + A[i];
}
```

2.6 Functions

A number of functions are provided in ArrayOps, that take arrays as function-arguments. Many of these functions are simply array-versions of their scalar counterparts, while others are proprietary for ArrayOps.

2.6.1 Mathematical Functions

The following mathematical functions are available for arrays: `abs`, `fabs`, `ceil`, `floor`, `sqrt`, `pow`, `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `atan2`, `cosh`, `sinh`, `tanh`, `exp`, `log`, `log10`, and `fmod`. The functions map directly to the functions from the `std`-namespace, and the argument- and return-types of your arrays must therefore be appropriate.

These functions are evaluated as any other mathematical operator, meaning that they are only evaluated if the expression in which they occur, is actually assigned to an array.

Example

Say we wish to compute:

$$C = \sin(A) + \sqrt{\cos(B)}$$

for some arrays $A, B, C \in \mathbb{R}^k$, which means we wish to compute:

$$C_i = \sin(A_i) + \sqrt{\cos(B_i)},\ \forall i \in \{0, \cdots, k-1\}$$

As usual, we may use the floating-point datatype `double` to approximate real-values, and using ArrayOps for the arrays of length, say $k = 1024$, we would have the following source-code to compute the expression:

```
const unsigned int k = 1024;           // Array size.
Array<double> A(k), B(k), C(k);        // Arrays.

// ...

C = sin(A) + sqrt(cos(B));             // The expression.
```

The last expression is automatically transformed by the ArrayOps meta-programming framework, to the following loop:

```
for (unsigned int i=0; i<C.Size(); i++)
{
    C[i] = std::sin(A[i]) + std::sqrt(std::cos(B[i]));
}
```

Again, this transformation occurs internally in the compiler, and does not manifest itself in your source-code.

2.6.2 Power Functions

Power functions where the power k is an integer, may be computed by applying the multiplication operator a total of $\log_2(k)$ times, which may be advantageous over using the standard `pow`-function, or even manually writing the expression k times. The following specialized power-functions are available for all arrays if only the array elements support the multiplication function, and the functions are used as any other unary function: `pow2`, `pow4`, and `pow8`.

Example

Say we wish to compute:

$$C = A^2 + B^4$$

for some arrays $A, B, C \in \mathbb{R}^k$, which means we wish to compute:

$$C_i = (A_i)^2 + (B_i)^4, \forall i \in \{0, \cdots, k-1\}$$

As usual, we may use the floating-point datatype `double` to approximate real-values, and using ArrayOps for the arrays of length, say $k = 1024$, we would have the following source-code to compute the expression:

```
const unsigned int k = 1024;           // Array size.
Array<double> A(k), B(k), C(k);        // Arrays.

// ...

C = pow2(A) + pow4(B);                 // The expression.
```

The last expression is effectively transformed by the ArrayOps meta-programming framework, into the following loop:

```
for (unsigned int i=0; i<C.Size(); i++)
{
    double A1 = A[i];
    double A2 = A1 * A1;

    double B1 = B[i];
    double B2 = B1 * B1;
    double B4 = B2 * B2;

    C[i] = A2 + B4;
}
```

A clear advantage to this, is that you may have more complicated arithmetic expressions, such as:

```
C = pow8(pow2(cos(A)) + pow2(sin(B)));
```

which compiles into the following:

```
for (unsigned int i=0; i<C.Size(); i++)
{
    double A1 = std::cos(A[i]);        // A2 = pow2(cos(A[i]));
    double A2 = A1 * A1;

    double B1 = std::sin(B[i]);        // B2 = pow2(sin(B[i]));
    double B2 = B1 * B1;

    double R1 = A2 + B2;               // R8 = pow8(A2 + B2);
    double R2 = R1 * R1;
    double R4 = R2 * R2;
    double R8 = R4 * R4;

    C[i] = R8;
}
```

where we have used as few calculations as possible for the given expression.

2.6.3 Size

Recall that the `ArrayBase`-class is exactly used to define those array-types that may occur as left-hands in assignments, and these array-types must have well-defined sizes. The following functions are provided to query the size of such an array:

- `bool IsSized()`, which must always return `true` for an instance of the `ArrayBase`-class.
- `unsigned int Size()`, which returns the size of the array (possibly zero).

These functions are actually available for all expressions involving arrays, so you may also call the functions in the following manner:

```
unsigned int n = (A+B).Size();
```

where `A` and `B` are some arrays.

2.6.4 Eval

A number of functions are provided, that let you use your own functions and functor objects in expressions involving arrays. It was considered whether or not to allow non-const functor objects in such expressions, but it was finally decided, that it was safer to disallow side-effects altogether, in arithmetic expressions involving arrays. (See section 2.8.4 for more details on this.)

Unary Functions

A couple of functions are provided for supporting the user's own unary functor objects, as well as pointers to unary functions. In all of these functions,

the template-argument `T` refers to the type of the elements in the arrays, the template-argument `F` is the functor-class, and the template-argument `S` is used internally by the ArrayOps framework, and you should not have to worry about it, as it is automatically deduced by the C++ compiler.

The functions are documented here with just their template-arguments and the function header. The return-value of the functions are internal ArrayOps-objects, which are nonsensical for someone who does not understand the inner-workings of the ArrayOps implementation (see chapter 3 for this). The functions are as follows:

- `template <typename T, class F, class S>`
  `Eval1 (Expr<T, S> const& expr, F const& f)`
  This function takes an array-expression as argument along with an instance of the functor. Note the `const`-ness of both.
- `template <class F, typename T, class S>`
  `Eval1 (Expr<T, S> const& expr)`
  This function takes an array-expression as argument, but makes an instance of the functor `F` by itself.
- `template <typename T, class S>`
  `Eval1 (Expr<T, S> const& expr, T (*f)(T))`
  This function takes an array-expression as argument along with a pointer to a function `f`; for which C++ does not make it possible to ensure const-ness.

Note that the ordering of the template arguments, are not the same for all of these functions, so as to easen their automatic deductions by the compiler. The functions are also available with different argument- and return-types, where the template-argument `T` is then replaced by `TArg` and `TRes`:

- `template <typename TRes, typename TArg, class F, class S>`
  `Eval1 (Expr<TArg, S> const& expr, F const& f)`
- `template <class F, typename TRes, typename TArg, class S>`
  `Eval1 (Expr<TArg, S> const& expr)`
- `template <typename TRes, typename TArg, class S>`
  `Eval1 (Expr<TArg, S> const& expr, TRes (*f)(TArg))`

Example

Let us say we are given arrays A, B, and C of length k, and of some arbitrary type, and we wish to compute the following expression:

$$C = A + B^8$$

which is here taken to mean:

$$C_i = A_i + B_i^8, \forall i \in \{0, \cdots, k-1\}$$

Suppose we did not have the eigth'-power functionality readily available in ArrayOps, then we could either use the mathematical `pow`-function from above, or we would have to write out `B*B*...*B` eight times. Alternatively however, we may implement the eight'-power functionality as a functor-class in the following manner:

```
template <class T>
class Pow8 : public std::unary_function<T, T>
{
public:
    Pow8() : std::unary_function<T, T>() {}

    inline
    T operator() (T const& x) const
    {
        T x2 = x*x;
        T x4 = x2*x2;
        T x8 = x4*x4;

        return x8;
    }
};
```

Then we may compute the mathematical expression $C = A + B^8$ as follows, using ArrayOps with one of its `Eval1` functions:

```
const unsigned int k = 1024;           // Array size.
Array<double> A(k), B(k), C(k);        // Arrays.
Pow8<double> f;                        // The functor

// ...

C = A + Eval1(B, f);                   // The expression.
```

Where the last expression is automatically transformed by the ArrayOps meta-programming framework, into the following loop:

```
for (unsigned int i=0; i<C.Size(); i++)
{
    C[i] = A[i] + f(B[i]);
}
```

2.6.5 Casting

ArrayOps was chosen to be strong-typed, which means that even the implicit C++ type-promotions are disallowed. The four usual casting functions are provided, but are named differently, so as to still allow the use of the traditional casting on sub-classes of the array-types. The casting functions in ArrayOps are as follows:

- `StaticCast` maps to `std::static_cast`.
- `DynamicCast` maps to `std::dynamic_cast`.
- `ConstCast` maps to `std::const_cast`.
- `ReinterpretCast` maps to `std::reinterpret_cast`.

You may not find all of these casting-functions useful; in fact, you are probably only ever going to need the `StaticCast`-function.

Example

Let us say we wish to add the `int`-array `A` to a `float`-array `B`, and finally assign this sum to the `double`-array `C`. The ArrayOps source-code for doing this, would be as follows:

```
const unsigned int k = 1024;           // Array size.
Array<int> A(k);                       // Array A.
Array<float> B(k);                     // Array B.
Array<double> C(k);                    // Array C.

// ...

C = StaticCast<double>(StaticCast<float>(A) + B);
```

Where the last expression is automatically transformed into the following loop:

```
for (unsigned int i=0; i<C.Size(); i++)
{
    C[i] = std::static_cast<double>(
           std::static_cast<float>(A[i]) + B[i] );
}
```

2.6.6 ReduceAll

A number of functions are provided for reducing the contents of an array into a single scalar value. The idea is essentially to use a unary functor for accumulation, and then traverse the array by calling the functor with each of the array's elements. For example, if `f` is such a functor, we want to perform the following sequence of computations on it:

```
// Initialize functor f ...

const unsigned int kSize = x.Size();

for (unsigned int i=0; i<kSize; i++)
{
    f(x[i]);
```

```
}

// Retrieve result from functor f ...
```

You can imagine the result if the functor `f` is initialized internally with a value of zero, and if the unary function invocation really performs the operation `+=` on the internal value with the current argument. Exactly, this is basic summation!

This functionality is provided by the following set of functions all named `ReduceAll`, but taking different template- and function-arguments, depending on whether the accumulation functor is provided or must be instantiated by the `ReduceAll`-function, and whether the argument- and return-types are identical. First are the functions where these types are the same:

- `template <class F, class T, class S>`
  `T ReduceAll(Expr<T, S> const& x, F& f)`, which has the functor provided as an argument. Note that the functor is of course not passed as a `const`-reference, as it would not be able to update its contents then.
- `template <class F, class T, class S>`
  `T ReduceAll(Expr<T, S> const& x)`, which instantiates its own functor.

The functions are also available with different argument- and return-types, where the template-argument `T` is then replaced by `TArg` and `TRes`:

- `template <class TRes, class TArg, class F, class S>`
  `TRes ReduceAll(Expr<TArg, S> const& x, F& f)`
- `template <class F, class TRes, class TArg, class S>`
  `TRes ReduceAll(Expr<TArg, S> const& x)`

In all cases, it is assumed that the functor-class `F` provides two functions, here written for the case where the argument-type `TArg` differs from the return-type `TRes`. In case they are identical, we would of course just write one common type `T` instead:

- `void operator() (TArg const& x)`, which is a unary function taking as argument the value `x` that must be accumulated.
- `TRes opperator() ()`, which is a nullary function that simply returns the result of the entire accumulation.

The reason that you must provide two functions, is that the ultimate querying might also perform some calculations that are unnecessary during the accumulation stage, and we thus avoid this overhead by only performing those calculations once at the very end.

Note that we prefer accumulative operators, as we thereby avoid the need for temporary values, and for more complicated user-defined types, accumulative operators are often cheaper to execute.

Accumulator-Class

ArrayOps has a basic implementation of such an accumulator where the argument- and return-types are identical:

```
template <typename T, class F>
class Accumulator : public std::unary_function<T,T>
{
public:
    Accumulator(T const& init) :
         std::unary_function<T,T>(),
         mFunctor(), mAcc(init) {}

    inline
    T const& operator() (T const& x)
    {
        mFunctor(mAcc, x);
        return mAcc;
    }

    inline
    T const& operator() ()
    {
        return mAcc;
    }

    protected:
        const F       mFunctor;
        T             mAcc;
};
```

Note that an initial value must be provided in the class' constructor. This initial value is for example zero for computing the sum and one for computing the product. Also, the template argument `F` is supposed to be a simple functor merely wrapping an accumulative and binary operator, an example could be the following functor wrapping the `+=` operator:

```
template<class T>
struct assign_plus : public std::binary_function<T, T, T>
{
    inline
    T& operator()(T& l, T const& r) const
    {
        return (l += r);
    }
};
```

Note that we return `l` as a reference and not a value, as the operator `+=` is (presumably) accumulative, and hence does not produce a temporary value for its result.

Example

The summation function in ArrayOps as described in section 2.7.1 below, is actually just implemented as follows:

```
template <typename T, class S> inline
T Sum(Expr<T,S> const& x)
{
    return ReduceAll(x, Accumulator<T, assign_plus<T> >(0));
};
```

Initializing With First Array Element

This way of computing a reduction, such as the sum above, uses one more computation than strictly necessary. If we had instead initialized the accumulator with the first element of the array, then a single addition could be saved. This however, requires a conditional statement for the case when the array has no elements, in which case the result would be zero. So to save a single additive computation, we would not only need more complicated source-code, but the computational cost could possibly even increase, as branches are sometimes more expensive than arithmetic computations.

2.7 Reductions

A number of functions are provided for computing reductions, that is, functions that take as argument an array and return a scalar value. There are essentially two kinds of reductions in ArrayOps: Reductions that end up returning a value of the same type as that of the elements of the provided array; an example of which could be summation. And then there are reductions that return a value of some other type, such as the computation of the average value of an array's elements, which usually is supposed to return a floating-point value.

2.7.1 Sum

Let A be an array of length n, then the sum of its elements is:[1]

$$\text{sum}(A) = \sum_{i=0}^{n-1} A_i = A_0 + A_1 + \cdots + A_{n-1}$$

The ArrayOps function for computing the sum of an array's elements is:

[1] Arrays in mathematical notation are also indexed from zero up to their length minus one, to keep with the C++ notation.

- `template <typename T, class S>`
  `T Sum(Expr<T,S> const& x)`, which takes as argument any ArrayOps expression, and returns the accumulated sum of its elements.

Note that the template arguments are automatically deduced by the C++ compiler.

Example

Since we may pass any ArrayOps expression to the summation function, we could have something like:

```
const unsigned int k = 1024;           // Array size.
Array<int>     A(k), B(k);             // Arrays.

// ...

int s1 = Sum(A);
int s2 = Sum(A*B);
```

Where `s1` holds the sum of the elements in array `A`, and `s2` holds the sum of the elements in the expression `A*B`. Note that we do not actually compute a temporary array for storing the result of the expression `A*B`, but the summation function merely evaluates the expression for each index, thus computing and accumulating the elements resulting from the arithmetic expression, one at a time.

2.7.2 Product

Let A be an array of length n, then its product is:

$$\text{prod}(A) = \prod_{i=0}^{n-1} A_i = A_0 \cdot A_1 \cdots A_{n-1}$$

The ArrayOps function for computing this product is:

- `template <typename T, class S>`
  `T Product(Expr<T,S> const& x)`, which takes as argument any ArrayOps expression, and returns the accumulated product of its elements.

Again, the template arguments are automatically deduced by the C++ compiler.

2.7.3 Mean

Let A be an array of length n, then the arithmetic mean is defined as:

$$\text{mean}(A) = \frac{1}{n} \sum_{i=0}^{n-1} A_i$$

There are two ways of computing the mean of an array's elements in ArrayOps, the first is by calling the following function:

- `template <typename T, class S>`
  `double Mean(Expr<T,S> const& x)`, which returns the mean of the array `x` as a `double`-typed value.

Here, the template arguments are automatically deduced by the C++ compiler, but if you do not want the return-value to be `double`-typed, then you must call the static `Mean()`-function inside the `Reduce`-class instead (see example below).

The reason that the `Reduce`-class is needed, is because template *functions* do not allow for default parameters, so we can not just simply introduce another template argument `TRes` and let it default to `double`, however convenient it would have been.

Example

Let us compute the mean of various arrays and arithmetic expressions involving those arrays:

```
const unsigned int k = 1024;           // Array size.
Array<int>     A(k), B(k);             // Arrays, int.
Array<double>  C(k);                   // Array,  double.
Array<float>   D(k), E(k);             // Arrays, float.

// ...

double m1 = Mean(A);
double m2 = Mean(A*B);
double m3 = Mean(C);

float  m4 = Reduce<float>::Mean(D/E);
```

The first few computations of the mean values, all result in a `double`-typed return-value. The last results in a `float`-typed return-value. This example is perhaps a bit contrived, and usually you will only want to use the `Reduce`-class if you have some user-defined datatype, such as an arbitrary-precision fixed-point numeric type.

2.7.4 Norm

The Euclidian (or L_2) norm of an array A of length n, is defined as:

$$\text{norm}(A) = \sqrt{\sum_{i=0}^{n-1} A_i^2}$$

Again, there are two ways of computing the norm of an array's elements in ArrayOps, the first is by calling the following function:

- `template <typename T, class S>`
  `double Norm(Expr<T,S> const& x)`, which returns the norm of the array `x` as a `double`-typed value.

If you do not want the return-value to be `double`-typed, then you must call the static `Norm()`-function inside the `Reduce`-class (see example for the `Mean()`-function above).

2.7.5 Variance

The population variance for an array A of length n is defined as follows:

$$\mathrm{var}(A) = \frac{1}{n}\sum_{i=0}^{n-1}\left(A_i - \mathrm{mean}(A)\right)^2$$

Again, there are two ways of computing the variance of an array's elements in ArrayOps, the first is by calling the following function:

- `template <typename T, class S>`
  `double Variance(Expr<T,S> const& x)`, which returns the variance of the array `x` as a `double`-typed value.

If you do not want the return-value to be `double`-typed, then you must call the static `Variance()`-function inside the `Reduce`-class (see example for the `Mean()`-function above).

2.8 Semantics

ArrayOps was chosen to be fairly rigid in its semantic rules. Sometimes there was no choice, due to the way ArrayOps is implemented, and at other times, rigid semantics were chosen as they seemed to make most sense. This section describes the semantic rules, and what ArrayOps does to help you obey them.

2.8.1 Implicit Resizing

An important aspect of ArrayOps, is that none of its array-types have implicit resizing and allocation of storage – unless otherwise explicitly noted, as for example with the `ArrayAuto` class from page 12. This means that you should be able to use them in time-critical applications, at interrupt-level, etc. This also means that no resources are spent during execution, to check whether an array needs resizing.

Typically, you allocate the arrays at the beginning of some function, and then perform hundreds, thousands, or maybe even thousands of thousands (they call them millions!) of iterations, that evaluate some arithmetic expression on the arrays. Having implicit `if`-statements checking the size on each of these iterations, when you already know that the size is correct, is of course redundant.

2.8.2 Strong-Typed

ArrayOps is strong-typed, even more so than the C++ programming language itself. This means that arrays may only be combined in arithmetic expressions, if they are of matching type, and implicit type-promotions are disallowed. If you should try and combine arrays with elements of different datatypes, the compiler should halt with an error.

This is really due to the way ArrayOps is implemented; but it is not really a limitation, but rather an assistance to the user of ArrayOps. The reason is that ArrayOps is of course a numeric library, intended for numeric computations, where it is important to have control over such things as rounding errors. Since ArrayOps does not allow implicit conversion between types, then the user of ArrayOps must explicitly decide how to convert between datatypes each time. This way, you avoid any surprises in erroneous conversions, unless of course, you make the bug yourself.

2.8.3 Size-Matching

In vein of the strong-typed semantics of ArrayOps, it was also chosen to only allow arrays with matching sizes to occur in arithmetic expressions.[2] Unfortunately, this may not be checked at compile-time, because the ArrayOps framework generally allows for runtime changes in array-sizes.

From the discussion in section 2.8.1, it seems natural that we do not wish to check arithmetic expressions, whether their sub-expressions and arrays are of matching size, in the final release-build of the software using ArrayOps, because ArrayOps is normally used in a fashion, where we would be able to test this during development. The array-sizes are therefore enforced to match, by using the `assert()`-function from C++, which only builds in the debug-version.

2.8.4 Constness

Another important aspect of ArrayOps, is that all arrays occuring in an arithmetic expression, must support `const` lookup-functions of its elements. This severely limits our abilities to create exciting and clever array-types, for example for random number generation, as that would have to update internal variables, and hence could not be `const`.

So why was this rule chosen; particularly when it is not strictly required implementation-wise? The reason is simply that side-effects are troublesome in more than one way. First off, arithmetic expressions with side-effects are much harder to understand and maintain, and since the ArrayOps-framework is already not the easiest thing to debug for a regular user, having side-effects in arithmetic expressions could quickly become disastrous.

Secondly, ArrayOps supports parallelism through multi-threading (see section 2.9), which requires for the arithmetic-expressions to either be computed

[2] The exception of course being the `ArrayAuto`-class which automatically resizes according to the right-hand of an assignment. See page 12.

thread-safely, or for the ArrayOps framework to determine whether a particular arithmetic expression is thread-safe or not, and only compute those that are thread-safe in parallel. It is actually possible to implement this kind of checking using template meta-programming, but it is not desirable as it increases the maintenance-difficulty considerably.

It is still possible to have side-effects in arithmetic expressions, although it is strongly discouraged (and the feature may change or be removed entirely in future versions). This is done using the `Eval`-function from section 2.6.4, that takes a pointer to a function. Such functions addressed through pointers may not be declared `const` in C++, and hence provides an opportunity for having side-effects in arithmetic expressions.

2.8.5 Assertions

Assertions are used whenever possible, to check whether array-indices and array-sizes are correct, to check if pointers are non-zero, and so on. Recall that assertions are not compiled in the release-build of your program, and therefore only raise exceptions in the debug-build of the program. This is both good and bad, depending on what kind of error-checking you are hoping for.

Using assertions is good because you may have an abundance of error-checking, which helps you during development to quickly pinpoint the source of an error. Since assertions are not built into the release-version of your program, this multitude of error-checking does not take up any resources in the final program. But this is also the weak point of course, since assertions thereby do nothing for you in terms of error-checking and error-handling in the release-build of your program. This must be done using traditional exceptions; that is, by using the `throw` and `catch` statements.

2.8.6 Exceptions

In the current version of ArrayOps, not much attention was paid to exceptions. As described in section 2.3.7, functions are provided for checked lookup, that throw exceptions if an index is out of bounds – also in the release-build.

But exceptions in arithmetic expressions are presently ignored in the ArrayOps implementation. That is, if some arithmetic expression should raise an exception, then ArrayOps does not catch and handle it; you the user are assumed to do that yourself.

2.9 Parallelism

Parallelism in ArrayOps is implemented using OpenMP [Board], which is easily implemented in the loop of the assignment operator.

2.9.1 ArrayBase Support

Each instance of the `ArrayBase`-class is designated as either being parallel or not. This is done through a boolean template-argument to the class, which defaults to either `true` or `false` for the different array-types (see section 2.3), and which may be changed by the user upon instantiation of an array.

Furthermore, the `ArrayBase`-class defines two functions for setting a parallel limit used by OpenMP, for deciding whether or not the arrays are big enough to merit the use of parallelism. This parallel limit may be changed at will, and the functions for setting and querying the limit are as follows:

- `int GetParLimit()`, that returns the limit, below which execution must be performed sequentially.
- `void SetParLimit(int parLimit)`, that sets the limit, below which execution must be performed sequentially.

Note that OpenMP only works with signed integers, which is also the cause of the type-casting of the array-sizes in the examples below.

Example

Recall that the `Array`-class defaults to having parallel execution enabled. So in the following example, whenever arrays `A` and `B` occur as the left-hand of an assignment, the corresponding loop will be performed in parallel:

```
const unsigned int k = 1024;          // Array-size.
Array<int> A(k), B(k);                // Arrays A, B.
Array<int, false> C(k);               // Array C.

// ...

A = B + C;                            // Parallel.
```

Where the assignment `A = B + C;` is automatically transformed into the following loop by the ArrayOps framework:

```
const int kSize = (int) A.Size()

#pragma omp parallel for if (kSize>=A.GetParLimit())
for (int i=0; i<kSize; i++)
{
    A[i] = B[i] + C[i];
}
```

On the other hand, if the assignment had instead been the following:

```
C = A + B;                            // Non-parallel.
```

where `C` was defined in the above to be non-parallel, then the expression would be transformed into the more familiar sequential loop:

```
const int kSize = (int) C.Size()

for (int i=0; i<kSize; i++)
{
    C[i] = A[i] + B[i];
}
```

Since this boolean for deciding whether to use parallelism or not, is given as a template argument, we may at compile-time, decide whichever loop is the appropriate.

2.9.2 Cache Coherency

An important issue when performing parallel programming using OpenMP, is that of cache coherency. OpenMP is based on a socalled *Shared Memory Processor* (SMP) architecture, in which multiple processors share their memory. Each processor has several layers of cache to increase performance in accessing the shared memory, but if one processor should update the memory that is held in one or more of the other processors' cache, then a penalty is incurred in the execution time.

However, because the right-hand of an ArrayOps assignment has no side-effects, and the left-hand is generally an array which should not occur in the right-hand of an assignment in non-trivial ways[3], the data that is written to the shared memory, is generally located far from the data that is read, and hence there should (generally) not be any problems with cache coherency, when using the OpenMP-based parallel execution in ArrayOps.

[3]For example, you should not have slices of the left-hand array appear in the right-hand of the assignment as well, as this is generally ill-defined if you do not know the execution order.

Chapter 3

Implementation

This chapter first describes the programming techniques used in the implementation of ArrayOps, and then documents the implementation itself. However, documentation of the macros – which is an essential part of the implementation – has presently been omitted, because the macros are likely to change, and should not be used by the user in their current form. Also, the actual implementation may differ slightly from what is described here, so as to better focus on the essentials, and hopefully make the implementation more comprehensible.

3.1 Techniques

The following sections are introductory descriptions of the programming techniques that are used in the implementation of ArrayOps.

3.1.1 Template Classes

Recall the concept of template classes in C++, in which you may use arbitrary types for their template arguments. Take for example a non-template class that sums a sequence of integers:

```
class Adder
{
public:
    Adder(): mSum(0) {}

    int operator() (int x) { return mSum += x; }

protected:
    int mSum;
};
```

Which may be instantiated and used as follows:

```
Adder sum;        // Object instance

sum(1);           // sum is 1
sum(2);           // sum is 3
sum(3);           // sum is 6
sum(4);           // sum is 10
```

If we wish to re-use this class for sums of numbers that are not necessarily integer-typed, then we may provide the type as a template argument as follows:

```
template <class T>
class Adder
{
public:
    Adder(): mSum(0) {}

    T operator() (T x) { return mSum += x; }

protected:
    T mSum;
};
```

Now we could also sum floating point numbers, by providing the appropriate type as a template argument to the `Adder`-class, for example as follows:

```
Adder<double> sum;  // Object instance

sum(1.1);           // sum is 1.1
sum(2.2);           // sum is 3.3
sum(3.3);           // sum is 6.6
sum(4.4);           // sum is 10.10
```

It is also possible to have other template arguments than just datatypes, for example one could have booleans or integers as template arguments. This is sometimes used in ArrayOps; just take the `ArrayMini`-class from section 2.3.3, where the user must provide the size of the array as a template argument (an unsigned integer).

Generating Specialized Code

The thing to remember about template arguments, is that they must always be provided at compile-time. This means the compiler will generate specialized code for that particular combination of template arguments.

Header & Source-File Separation

Another important thing, is that compilers presently do not allow for separation of template classes into header- and source-files. Everything must be provided

in the header-files, though you can still organize the implementation in smaller chunks in the header-file if you should find that desirable. This is not that bad for ArrayOps though, as mostly everything is implemented as template classes, because it also means that the user of the source-code library does not have to build any code-library or source-files.

3.1.2 Temporary Objects

The automatic and implicit construction and destruction of temporary objects in C++, is sometimes the source of vast amounts of overhead in the execution-time, or as described in [Stroustrup, 1991, Section 10.4.10, page 254]:

> Temporary objects most often are the result of arithmetic expressions. For example, at some point in the evaluation of `x*y+z` the partial result `x*y` must exist somewhere. Except when performance is the issue, temporary objects rarely become the concern of the programmer. However, it happens.

Temporaries are the reason why only few people (if any) use the `valarray` class from the Standard Template Library (STL). There are ways to get around the implicit usage of temporaries however, for example by introducing abstract objects instead[1] – or indeed, by using meta-programming as in ArrayOps.

Passing By Value & Reference

The regular C++ programmer does not deal in those kinds of things, but still, it is important that everyone knows when to pass and return values, and when to use references, so as to avoid the generation of temporary objects – particularly for template classes. For example, something as seemingly innocent as passing-by-value to a function, may be extremely expensive, if that particular datatype has a constructor which does much work (such as copying a large amounts of data), and this was not conceived during the development of the template class or function.

Take for example the summation-class from section 3.1.1 above, where the following function uses pass- and return-by-value:

```
T operator() (T x) { return mSum += x; }
```

Instead, we should really have implemented the function as follows:

```
T const& operator() (T const& x) { return mSum += x; }
```

where both the argument `x` is passed as a `const`-reference, and the summation variable `mSum` is returned as a `const`-reference. This ensures that for larger, non-basic datatypes, we would still have an efficient summation function, which did not spend any additional time creating temporary objects.

[1] But this introduces other problematic issues.

Lifetime of Temporaries

Another important issue with temporaries, is *when* the deletion of a temporary object occurs. The lifetime of a temporary object is also described in [Stroustrup, 1991, Section 10.4.10, page 254], and we quote here:

> Unless bound to a reference or used to initialize a named object, a temporary object is destroyed at the end of the full expression in which it was created. A *full expression* is an expression that is not a subexpression of some other expression.

So what was a full expression again? To understand this we must look at the C++ syntax, but for our current purpose, let it suffice to say, that the temporary object is deleted at the end of the statement in which it occurs. This means that we can not, say, have created a temporary object in a function, and then directly or indirectly return a reference to it. This is most important in the ArrayOps implementation, as you will see below.

3.1.3 Meta-Programming

An important aspect of templates, is that they may be specialized in the source-code also. This was originally intended to be used in specializing implementations for certain datatypes, for example to decrease the storage requirements for the `std::vector` class, when used to hold boolean arguments. But consider the following template function, for computing the sum of the elements in a basic C++ array:

```
template <unsigned int k> inline
int Sum(int const* arr) { return arr[0]+Sum<k-1>(arr+1); }

template <> inline
int Sum<1>(int const* arr) { return arr[0]; }

template <> inline
int Sum<0>(int const* arr) { return 0; }
```

Where the template argument `k` designates the number of elements in the array that are left in the summation. We provide two termination conditions, one for the special case where `k` is one, and another case where `k` is zero. The latter should only occur if the user explicitly calls the function with zero as template argument.

The length of the array `k` must of course be known at compile-time, as it is provided as a template argument. This is an essential requirement for all template arguments, so the compiler may specialize the template already at compile-time. Take for example the following C++ code:

```
const unsigned int k = 16;
int arr[k];
```

```
// ...

int mySum = Sum<k>(arr);
```

Here the compiler will first try and instantiate `Sum<16>`, which in turn uses `Sum<k-1>`, that is `Sum<15>`. This in turn will make use of a call to the template function `Sum<14>`, and so on. So the compiler will actually generate the following code for the last line, due to the inlining:

```
int mySum = arr[0]+arr[1]+arr[2]+ ... + arr[15];
```

So in effect, the template function called `Sum`, will be flattened during compilation, and is therefore a way to avoid the traditional loop-style summation. This kind of flattening is somewhat analogous to the inner-workings of the ArrayOps framework.

3.1.4 Nested Meta-Programming

What if we wanted to create a summation function for arbitrary datatypes and not just integers? Well, this is where things get a bit tricky, due to restrictions in the C++ language. We first have to wrap the function in a template class which designates the datatype `T` (and any other template arguments that you may need – here we only need `T`), and then have the template functions reside inside that class:

```
template <class T>
class SumWrapper
{
public:
    template <unsigned int k> static inline
    T Sum(T const* arr) { return arr[0]+Sum<k-1>(arr+1); }

    template <> static inline
    T Sum<1>(T const* arr) { return arr[0]; }

    template <> static inline
    T Sum<0>(T const* arr) { return 0; }
};
```

Now, it would be a bit cumbersome to use this directly, as one would have to write something like:

```
int mySum = SumWrapper<int>::Sum<k>(arr);
```

Instead, C++ has a neat little feature (which is essential to the ArrayOps implementation), in that it may automatically deduce the datatypes of template arguments, from the function parameters to a template function (you may want to read this sentence again). To make this work, we therefore have to make a function as follows:

```
template <unsigned int k, class T> inline
T Sum(T const* arr) { return SumWrapper<T>::Sum<k>(arr); }
```

Here, we only have to provide the `k` parameter, as C++ is of course not able to automatically deduce it from the function parameters (`k` is the length of the array, which is not provided anywhere). So we could have something like:

```
const unsigned int k = 128;
double arr[k];

// ...

double mySum = Sum<k>(arr);
```

Where C++ would automatically figure out, that the datatype `T` is actually supposed to be `double`.

3.1.5 Reverse Inheritance

To specialize a class in C++, we often use socalled virtual functions. Let us say we have a class `A`, and wish to specialize its functions `f` and `g` in a sub-class `B`, and let us say the functions `f` and `g` must take integer-parameters, and also return integers. This may be implemented as follows:

```
class A
{
public:
    // ...

    virtual int f(int x) = 0;
    virtual int g(int x) = 0;
};

class B : public A
{
public:
    // ...

    virtual int f(int x) { /* return something */ }
    virtual int g(int x) { /* return something */ }
};
```

The trouble with the `virtual`-construct, is that it generally has a small overhead in execution time (there are exceptions to this rule). If we were to execute the `f` and `g` functions many many times, and the functions themselves were quite inexpensive, then the overhead would become a substantial part of the execution time.

Another solution is to use socalled reverse inheritance, which has the benefit that we may still say that class `B` is a sub-class of class `A`, but where the functions to be specialized may be declared `inline`, thus avoiding the overhead of making them `virtual`. The main drawback of this technique, is that the class-hierarchy is no longer that obvious for the human eye, so the technique should only be used sparingly and when absolutely necessary. Once again we make use of template arguments, and this time we provide the specialization in a super-class of `A` instead:

```
template <class S>
class A : public S
{
public:
    // ...

    // Assume super-class S provides functions f and g.
};

// The implementor-class for class B
class B_Imp
{
public:
    // ...

    inline int f(int x) { /* return something */ }
    inline int g(int x) { /* return something */ }
};

// Convenient type-definition for the class B.
typedef A<B_imp> B;
```

Note that the super-class `B_Imp` has functions `f` and `g` declared as `inline`. So once the user instantiates class `B` (which is really class `A<B_Imp>`), the function `f` may be called on that object, and this will be compiled as an inline function, and thus avoids the use of a lookup-table for virtual functions, which imposes a significant overhead for small and frequently used functions; as discussed above.

The technique of reverse inheritance is essential in ArrayOps, as it enables us to re-use the majority of the framework, by providing a single class named `Expr` for an arithmetic expression, and make various specializations of this class. Because of reverse inheritance, the functions of these specializations may be declared `inline`, while the different kinds of expressions are still sub-classes of the `Expr`-class. The operator-overloadings are then made as template functions that automatically deduce the super-class `S`, as described in section 3.1.4 above.

Initialization

Suppose that we wanted to initialize `B_Imp` with some parameters:

```
// The implementor-class for class B
class B_Imp
{
public:
    B_Imp(int i, char c) : kI(i), kC(c) { /* ... */ }

    inline int f(int x) { /* return something */ }
    inline int g(int x) { /* return something */ }

protected:
    const int  kI;
    const char kC;
};
```

The trouble here, is that class `B` can not initialize the member-fields of its (ultimate) super-class `B_Imp`, due to restrictions in the C++ language. Another solution would be to pass the parameters `i` and `c` to the constructor of class `A`, and pass them further on to `A`'s super-class. But this only works if class `A` is always used with super-classes that require these specific parameters – which then looses much of the versatility of the reverse inheritance technique.

Instead we pass an instance of `B_Imp` to the constructor of class `A`, which in turn passes it to its super-class as well. This means that we call a constructor of `B_Imp` with an instance of itself as argument. C++ automatically provides such a constructor for any class,[2] which copies all the member-fields.[3] So we may pass a `B_Imp`-object through class `A` as follows:

```
template <class S>
class A : public S
{
public:
    A (S const& s) : S(s) { /* ... */ }

    // Assume super-class S provides functions f and g.
};
```

And then we can make a new class `B` that takes the appropriate arguments, and sends a temporary instance of `B_Imp` to the constructor of `A`, which in turn sends it to its super-class `B_Imp`, which then copies the member-fields by default:

```
class B : public A<B_imp>
{
public:
    B(int i, char c) : A<B_imp>(B_Imp(i, c)) { /* ... */ }
};
```

[2] Unless we explicitly override it.

[3] If we require some special kind of initialization code to be performed, then we could make a constructor as follows: `B_Imp(B_Imp const& x) : kI(x.kI), kC(x.kC) { /* ... */ }`

Since the instance we create by calling `B_Imp(i, C)` is temporary, it is important to remember, that it is destroyed once the call to the constructor of class `A` returns. So you should generally not perform expensive calculations or allocations in the constructor and destructor of `B_Imp`, as they would then be called twice. Instead, you should provide this functionality separately, and call it from the constructor and destructor of class `B`.

3.1.6 Macros

Much of the ArrayOps implementation re-uses the same textual source-code with only small modifications. To do this efficiently, one can use macros. However, macros are not very powerful, and one problem with macros that is particularly annoying in ArrayOps, is that arguments to macros can not have commas in them, as the comma is used to separate the macro-arguments themselves.

3.2 Framework

The basic idea in the meta-programming framework of ArrayOps, is to overload the various operators (the functions `operator+`, `operator*`, and so on), and have them build and return an expression tree. This is done at compile-time using template classes, that mimics the mathematical expression you have written in your source-code, and the expression tree will then be flattened to a single loop, due to inlining of the lookup functions of each sub-expression. Again, since the template parameters of these classes are provided at compile-time, the code is specialized and optimized at compile-time.

3.2.1 Class Hierarchy

The ArrayOps implementation has a single class named `Expr`, that all mathematical expressions involing arrays must derive from. This gives us the ability, to clearly tell whether an object is an expression involving arrays, and thus enables us to re-use the entire framework, for all variants of the `Expr`-class.

To do this efficiently, we must always split such classes in two: The implementation that will serve as super-class to the `Expr`-class through reverse inheritance, and the actual class that will be instantiated.

3.2.2 Functors

The ArrayOps implementation uses socalled functors to provide and apply functions to mathematical expressions. This means that we always have to wrap, for example, static functions in appropriate functor objects.

3.2.3 Storage-Class

Before we start describing the meta-programming framework for the ArrayOps implementation, let us first give a class used for storing data of arbitrary types.

It will not become clear why this class is actually necessary until sections 3.2.7 and 3.3, but since it is used in almost every class of the meta-programming framework, we describe it here, and for the time being, let it suffice to say, that it makes the ArrayOps framework much simpler in the long run.

Storage Abstraction

For storing either a reference or a copy of some data, we provide the following abstract class:

```
template <class X, bool Copy>
class Storage
{
};
```

Where `X` is the datatype for the object that must be stored, and `Copy` designates whether to store a copy of the object, or just a reference to that object. When we say copy, we really mean that we store an actual instance, that may have its contents copied during intialization of the `Storage`-object (more on this later).

The `Storage`-class is specialized depending on the value of the `Copy` template boolean argument. It is furthermore assumed that each specialization implements the following constructor and function:

- `Storage(X const & x)`, which is the constructor that takes the object to be referenced or copied as parameter.
- `X const& Get() const`, which is the function for retrieving the data.

The user of the `Storage`-class is of course assumed to ensure that data stored only as a reference, is still available once retrieved.

Storing A Reference

First is the specialization of the `Storage`-class where we should not copy the data that we wish to store, but merely hold a reference to it. This is the case when the `Copy` template argument has the value `false`:

```
template <class X>
class Storage<X, false>
{
public:
    Storage(X const& x) : mX(x) {}

    inline X const& Get() const { return mX; }

protected:
    X const& mX;
};
```

Note that the referenced data is declared `const`, meaning that we are not allowed to make any alterations to the data – this is due to the semantic principle in ArrayOps, that no side-effects are allowed in mathematical expressions.

Again, the user of the `Storage`-class is assumed to ensure that the data being referenced, is still available once retrieved through the `Get()`-function. The `Storage`-class would have no way of knowing if this data has been deleted elsewhere in the program.[4] In such a case where you either do not know if the data is still available, or if you are certain that it is not, you should use the `Storage`-specialization that copies the data instead of just holding a reference to it.

Storing A Copy

Next is the specialization of the `Storage`-class that stores an instance or a copy of the data, that is, where the `Copy` template argument has the value `true`:

```
template <class X>
class Storage<X, true>
{
public:
    Storage(X const& x) : mX(x) {}
    Storage() : mX() {}

    inline X const& Get() const { return mX; }

protected:
    X mX;
};
```

Where it is assumed that the class `X` has an appropriate constructor that copies its data.[5]

Note that an additional constructor which takes no arguments, has been supplied in this version of the `Storage`-class. This is used when one merely wants to hold an instance of class `X`, but not initialize it with any particular data – this is for example the case when using the `Storage`-class for instantiating operator-functors (see e.g. section 3.2.5 on page 51 below).

3.2.4 Expr-Class

The class which every mathematical expression must implement through reverse inheritance, is the `Expr`-class defined as follows:

```
template <typename T, class S>
class Expr : public S
```

[4] Unless one uses a more complicated allocation and deallocation scheme, where objects have reference-counters.

[5] This is provided by default in C++, unless explicitly overrided in the source-code.

```
{
public:
    Expr () : S() {}
};
```

Where `T` is the datatype of the expression's elements (e.g. `int` or `double`), and `S` is the super-class for the reverse inheritance. There are some assumptions about the functionality of the super-class `S`, and also about an instance of the `Expr`-class in general. These assumptions are depicted below.

Implementor & Instantiator

But first, to make things a bit easier, let us define our terminology. In particular, we split the implementation for each kind of mathematical expression into two classes:

- The implementor, which is the super-class `S` of the `Expr`-class. Among other things, the implementor provides inlined lookup-functions of an array's elements, and the size-queries described below.
- The instantiator, which is a sub-class of the `Expr`-class, setting the `S` super-class to the appropriate implementor-class, and is therefore merely used as an easier way to instantiate the complete class.

This splitting into two classes is of course due to the use of reverse inheritance, and will become more obvious shortly.

Size-Queries

An instance of the `Expr` class is always assumed to implement a lookup function by overriding the `operator[]`-function, as well as implementing the two following functions for querying the size of the expression:

- `bool IsSized()`, which returns whether an expression is sized or not. For example, the expression containing a constant value is not sized, whereas an array must be sized.[6]
- `unsigned int Size()`, which returns the size of the expression provided `IsSized()` returned `true`, otherwise the result of calling `Size()` is undefined.

Again, these size queries must be available for all instances of the `Expr`-class, so as a user of ArrayOps, you can assume the queries to be available for all mathematical expressions involving arrays.

The way these queries are implemented, is usually by direct traversal of the given expression tree. This however, should be optimized by the compiler to

[6] An array is also considered a mathematical expression in ArrayOps, since it implements the `Expr`-class.

a single inlined function-call requiring constant execution time, due to the fact that the leaves of the expression tree have their `IsSized()`-functions always returning the same value, and the unary and binary expressions have a simple relation between the `IsSized()`-query of their sub-expressions, and which sub-expression to perform the `Size()`-query on. Alternatively, one could compute and store the results of the size queries for a given expression, during the construction of an object in the meta-programming framework, but this would be much harder for the compiler to optimize.

Checked Lookup Function

The `Expr`-class provides the `const` checked lookup function that throws an exception if the index is out of range. The function merely uses the `operator[]` function that is assumed to be provided by the implementor-class supplied through the template argument `S`, and is hence implemented as follows:

```
// Element lookup, const.
T const& at(unsigned int i) const
{
    if (i >= S::Size())
    {
        throw std::out_of_range(kErrRange);
    }

    return (*this)[i];
}
```

The non-`const`-version of this function is declared in the `ArrayBase`-class (see page 73). The error-message for both these functions has been defined as:

```
const std::string kErrRange("Index out of range.");
```

3.2.5 Expr1-Class

Unary expressions are implemented by the `Expr1`-class. An example of a unary function, could be that of negation. Say you have some array `A` and wish to write `-A`, then this unary expression will be stored as an instance of the `Expr1`-class, in the meta-programming framework of ArrayOps.

Implementor

The implementor for the `Expr1`-class – that is, the class that will ultimately serve as the super-class for the `Expr`-class – is implemented as follows:

```
template <typename T, class Op, bool CopyOp,
                      class X,  bool CopyX>
class Expr1_Imp
{
```

```
public:
    Expr1_Imp() : mOp(), mX() {}
    Expr1_Imp(X const& x, Op const& op) : mOp(op), mX(x) {}

    inline
    T operator[] (unsigned int index) const
    {
        // Get the operator functor.
        Op const& op = mOp.Get();

        // Apply operator and return result.
        return op(mX.Get()[index]);
    };

    inline
    bool IsSized () const { return mX.Get().IsSized(); }

    inline
    unsigned int Size () const { return mX.Get().Size(); }

protected:
    Storage<Op, CopyOp> mOp;        // Operator functor.
    Storage<X, CopyX>   mX;         // Sub-expression.
};
```

Note that there is no non-`const` lookup-function, because it would be meaningless. Also note that the lookup-function and the size-queries are all declared `inline`.

Instantiator

Now comes the instantiator for the `Expr1`-class, that is, the class that we will use to actually instantiate such an object, instead of manually using the `Expr1_Imp`-class. The `Expr1`-class is as follows:

```
template <typename T, class Op, bool CopyOp,
                      class X,  bool CopyX>
class Expr1 : public Expr<T, Expr1_Imp<T, Op, CopyOp,
                                       X,  CopyX> >
{
public:
    // Convenient type-definition of implementor.
    typedef Expr1_Imp<T, Op, CopyOp, X, CopyX> TImp;

    Expr1(X const& x) : Expr<T, TImp>(TImp(x)) { }

    Expr1(X const& x, Op const& op) :
```

```
                    Expr<T, TImp>(TImp(x, op)) { }
};
```

Note how the constructors actually create temporary instances of the `Expr1_Imp`-class and pass them to the constructors of the `Expr`-class, which in turn pass the objects to the constructor of its own super-class, here `Expr1_Imp`. This may look a bit complicated at first, but once you get the hang of it, it simplifies the source-code a great deal, as opposed to setting pointers to the expression object `x` and operator functor `op` in the code-part of the constructors in the `Expr1`-class.

3.2.6 Expr2-Class

The class for holding binary expressions in the meta-programming framework is called `Expr2`, and is very similar to the class `Expr1` for unary expressions. An example of a binary function is addition of two sub-expressions. Say you have arrays `A` and `B` and wish to write `A+B`, then this binary expression, will be stored in an instance of the `Expr2`-class in the meta-programming framework of ArrayOps, with its left-hand sub-expression being `A`, its right-hand expression being `B`, and its operator functor being `std::plus`.

Implementor

The implementor for the `Expr2`-class – that is, the class that will ultimately serve as the super-class for the `Expr`-class – is implemented very similarly to the implementor for the `Expr1`-class, and is as follows:

```
template <typename T, class Op, bool CopyOp,
                      class L,  bool CopyL,
                      class R,  bool CopyR>
class Expr2_Imp
{
public:
    Expr2_Imp(L const& l, R const& r) : mOp(), mL(l), mR(r) {}

    Expr2_Imp(L const& l, R const& r, Op const& op) :
                                      mOp(op), mL(l), mR(r) {}

    inline
    T operator[] (unsigned int index) const
    {
        // Get the operator functor.
        Op const& op = mOp.Get();

        // Apply operator and return result.
        return op(mL.Get()[index], mR.Get()[index]);
    };
```

```
    inline
    bool IsSized () const
    {
        return (mL.Get().IsSized() || mR.Get().IsSized());
    }

    inline
    unsigned int Size () const
    {
        return (mL.Get().IsSized()) ? (mL.Get().Size())
                                    : (mR.Get().Size());
    }

protected:
    Storage<Op, CopyOp>  mOp;    // Operator functor.
    Storage<L, CopyL>    mL;     // Left operand.
    Storage<R, CopyR>    mR;     // Right operand.
};
```

As with the `Expr1_Imp`-class, note that there is no non-`const` lookup-function, because it would be meaningless. Also note that the lookup-function and size queries are all declared `inline`, and that the size queries *should* be optimized to a single overall call (see discussion on page 50).

Instantiator

Now comes the instantiator for the `Expr2`-class, that is, the class that we will use to actually instantiate such an object, instead of manually instantiate from the `Expr2_Imp`-class. The `Expr2`-class is as follows:

```
template <typename T, class Op, bool CopyOp,
                      class L,  bool CopyL,
                      class R,  bool CopyR>
class Expr2 : public Expr<T, Expr2_Imp<T, Op, CopyOp,
                                          L,  CopyL,
                                          R,  CopyR> >
{
public:
    // Convenient type-definition of implementor.
    typedef Expr2_Imp<T, Op, CopyOp, L, CopyL, R, CopyR> TImp;

    Expr2(L const& l, R const& r) : Expr<T, TImp>(TImp(l, r))
    {
        assert(MatchingSize(mL.Get(), mR.Get()));
    }
```

```
    Expr2(L const& l, R const& r, Op const& op) :
                                Expr<T, TImp>(TImp(l, r, op))
    {
        assert(MatchingSize(mL.Get(), mR.Get()));
    }
};
```

Again, note how these constructors create temporary instances of the `Expr2_Imp`-class and pass them to the constructors of the `Expr`-class, which in turn pass the objects to the constructor of its own super-class, here `Expr2_Imp`.

3.2.7 Values & Variables

Consider an expression like `A = b * B + 0.25 * C;` where `A`, `B`, and `C` are arrays with `double`-typed elements, and `b` is a `double`-typed variable, and `0.25` is a `double`-typed constant. Obviously, the constant exists somewhere in the program being compiled, and supposedly we could just as well use a reference to it, instead of copying it into a local variable when executing the actual assignment.[7] But since we are using meta-programming to duplicate the whole expression internally in the compiler, it turns out that we can not use a reference to the constant value `0.25`, as we can with the variable `b`. For this reason, we need expression-classes for both values and variables.

Implementor

The implementor class for both value- and variable-expressions is the same. Actually, the class is very much like the `Storage`-class from section 3.2.3, only we provide lookup through `operator[]` – where the index is just ignored – and implement the size-queries as well:

```
template <typename T, bool Copy>
class ExprValVar_Imp
{
public:
    ExprValVar_Imp(T const& value) : mValue(value) {}

    inline
    T const& operator[] (unsigned int index) const
    {
        return mValue.Get();
    };

    inline
    unsigned int Size () const { return 0; }
```

[7] For datatypes that are more expensive to copy, this makes sense.

```
    inline
    bool IsSized () const { return false; }

protected:
    Storage<T, Copy> mValue;
};
```

ExprVal Instantiator Class

When we have constant values in an expression involving arrays, such as the constant value `0.25` above, we have the following instantiator class, which merely ensures that the value is copied and not just referenced (note the `true`-valued template argument to the `ExprValVar_Imp`-class):

```
template <typename T>
class ExprVal : public Expr<T, ExprValVar_Imp<T, true> >
{
public:
    // Convenient type-definition of implementor.
    typedef ExprValVar_Imp<T, true> TImp;

    ExprVal(T const& value) : Expr<T, TImp>(TImp(value)) {}
};
```

Note that we use the same kind of initialization of the implementor-class, as we did for the `Expr1`- and `Expr2`-classes above.

ExprVar Instantiator Class

When we have variables in an expression involving arrays, such as the variable `b` in the example from page 55, we have the following instantiator class, which simply ensures that we store a reference to the variable (note the `false`-valued template argument to the `ExprValVar_Imp`-class):

```
template <typename T>
class ExprVar : public Expr<T, ExprValVar_Imp<T, false> >
{
public:
    // Convenient type-definition of implementor.
    typedef ExprValVar_Imp<T, false> TImp;

    ExprVar(T const& value) : Expr<T, TImp>(TImp(value)) {}
};
```

3.3 Operators

In ArrayOps, an operator is really just taken to mean a function whose argument- and return-types are identical. Currently, support for unary and binary opera-

tors are implemented, as they are the most common. Since ArrayOps does not allow for any side-effects, it is meaningless to support nullary operators, as they would always just produce the same value, and hence be identical to a constant value.

3.3.1 Unary Operator

An example of a unary operator is that of negation. The following piece of source-code implements negation for any kind of ArrayOps expression, whose elements are of any type. Recall that C++ will only actually instantiate this function if you use it, and you may therefore have a datatype `T` which does not support negation, but this function can still be declared, as long as you do not use it (if you do so anyway, you will ultimately get an error, because your datatype `T` does not implement `operator-`). The ArrayOps operator-overload for unary negation, is as follows:

```
template <typename T, class S>
Expr1<T, std::negate<T>, true, Expr<T, S>, false>
operator- (Expr<T, S> const& x)
{
    return Expr1<T, std::negate<T>, true,
                    Expr<T, S>, false>(x);
};
```

Note that we use the `std::negate`-functor, and pass a `true`-valued template argument to the `Expr1`-class, indicating the class should create and hold an object for this functor. The `false`-valued boolean template argument to `Expr1`, means that the parameter `x` should not be copied to the `Expr1`-instance that we are creating, but it should merely hold a reference to it. The reason for this, is that the expression tree is first built, then flattened, and finally also destroyed during compilation, and the expression `x` will survive long enough for us to use it for its purpose. Below we will see examples where this is not the case.

Also note that the parameter to the `operator-` overloading is declared `const`, which ensures that no side-effects are allowed.

3.3.2 Binary Operator

For binary operators – that is, operators that take two arguments and return just one – we need a series of functions, for the different cases where one of the arguments is a constant value or a variable. Recall that we have two kinds of expressions when dealing with constant values and variables. The expression for constant values stores a copy of the value, and the expression for variables, stores only a reference.

The C++ compiler automatically finds whichever operator overloading is most appropriate depending on whether or not the paramater is `const`. When the parameter is declared as a `const`-reference, then it most closely matches a

constant value, and when the parameter is declared as a non-`const`-reference, then it most closely matches a variable.

In the following we have exemplified this for the addition operator (whose functor is `std::plus`), but since the source-code would be identical for all operators, with the exception of the operator- and functor-names, this is actually implemented as macros in the ArrayOps framework.

Two Sub-Expressions

The most general case, is when both operands are expressions themselves. In this case, the operator-overloading becomes:

```
template <typename T, class S1, class S2>
Expr2<T, std::plus<T>, true,
         Expr<T, S1>, false,
         Expr<T, S2>, false>
operator+ (Expr<T, S1> const& l, Expr<T, S2> const& r)
{
    return Expr2<T, std::plus<T>, true,
                    Expr<T, S1>, false,
                    Expr<T, S2>, false>(l, r);
};
```

The objects for the expressions `l` and `r`, will live long enough for us to use them, so there is no need to copy them inside the `Expr2`-object that we are instantiating. This is indicated by the `false`-valued template arguments to the `Expr2`-class.

On the other hand, the `std::plus` functor object is not instantiated anywhere, unless we instruct the `Expr2`-class to do so. This is done by passing the appropriate `true`-valued template argument to the class, and this is the same for all the variants of the operator-overloading that now follow.

One Sub-Expression & One Constant Value

When either the left- or right-hand is not an expression but a constant value, then we have the following operator overloadings. First is the operator overloading if the right-hand is a constant value:

```
template <typename T, class S>
Expr2<T, std::plus<T>, true,
         Expr<T, S>, false,
         ExprVal<T>, true>
operator+ (Expr<T, S> const& l, T const& r)
{
    return Expr2<T, std::plus<T>, true,
                    Expr<T, S>, false,
                    ExprVal<T>, true>(l, ExprVal<T>(r));
};
```

And similarly the operator overloading for when the left-hand is a constant value:

```
template <typename T, class S>
Expr2<T, std::plus<T>, true,
         ExprVal<T>, true,
         Expr<T, S>, false>
operator+ (T const& l, Expr<T, S> const& r)
{
    return Expr2<T, std::plus<T>, true,
                    ExprVal<T>, true,
                    Expr<T, S>, false>(ExprVal<T>(l), r);
};
```

There are two things to note here, first that a constant value, say a value declared as `const double d = 10,5;` does not continue to exist until we will need it. Therefore we must copy the value, and this is done in the `ExprVal`-object.

The second thing to note, which is just as important, is that we create and send an `ExprVal`-object to the constructor of the `Expr2`-class. The trouble here is, that in the C++ object system, the object that we make with the statement `ExprVal<T>(r)`,[8] is actually destroyed once the `Expr2`-constructor returns. This is not so strange when you think about it, as the object (in view of C++ semantics) is only meant to be used as a parameter to the `Expr2` class' constructor. But it naturally means that we must ensure the object is also copied into the `Expr2`-object, which is what the `true`-valued boolean template arguments designate.

One Sub-Expression & One Variable

When either the left- or right-hand is not an expression but a scalar variable, then we have the following operator overloadings. First is the operator overloading if the right-hand is a variable:[9]

```
template <typename T, class S>
Expr2<T, std::plus<T>, true,
         Expr<T, S>, false,
         ExprVar<T>, true>
operator+ (Expr<T, S> const& l, T& r)
{
    return Expr2<T, std::plus<T>, true,
                    Expr<T, S>, false,
                    ExprVar<T>, true>(l, ExprVar<T>(r));
};
```

And the operator overloading for when the left-hand is a variable, is similar:

[8] And similarly for `ExprVal<T>(l)` of course.

[9] Note the difference from when the right-hand was a constant value, which was indicated by the `const` keyword.

```
template <typename T, class S>
Expr2<T, std::plus<T>, true,
         ExprVar<T>, true,
         Expr<T, S>, false>
operator+ (T& l, Expr<T, S> const& r)
{
    return Expr2<T, std::plus<T>, true,
                    ExprVar<T>, true,
                    Expr<T, S>, false>(ExprVar<T>(l), r);
};
```

Once again, there are two things to note, first that a variable, say a value declared as `double d;` does continue to exist until we need it. Therefore we do not need to copy the value, and may therefore use the `ExprVar`-class, which performs no such copying, but merely holds a reference.

The second thing to note however, is that we still must copy the `ExprVar` objects that we send to the `Expr2` constructor, because the `ExprVar` objects that we create, will be destroyed by the C++ system once the constructor for the `Expr2`-class returns. Again, this required copying is what the `true`-valued boolean template arguments to the `Expr2`-class designate.

3.4 Functions

Functions are provided for manipulating arrays; some of the functions do this one element at a time, and other functions manipulate all the elements in turn.

3.4.1 Eval1

The `Eval1` functions are used to apply your own functions to the elements of an array, one element at a time, and thus integrate seamlessly into the ArrayOps meta-programming framework. Since the ArrayOps framework uses functors, it is only natural that the user must provide the functionality to be applied, as a functor. The following sections list the implementations of the different variants of the `Eval1` functions, depending on what kind of functors are to be used. Note that the ordering of the template arguments differ for each of the `Eval1` functions, so as to improve the C++ compiler's ability to automatically deduce the template arguments.

Functor Object Provided

The first version of the `Eval1`-function takes a user-supplied functor object (that is, the `Eval1`-function should not instantiate its own functor object), and the argument- and return-types are identical. The function is as follows:

```
template <typename T, class F, class S>
Expr1<T, F, false, Expr<T, S>, false>
```

```
Eval1(Expr<T, S> const& expr, F const& f)
{
    return Expr1<T, F, false, Expr<T, S>, false>(expr, f);
};
```

Note that the C++ compiler automatically deduces the functor class `F`, and as we are returning an `Expr1` expression object, with the appropriate boolean template argument set to `false`, we do not copy the supplied functor object, but merely store a reference to it.

We may use this version of the `Eval1`-function in the following manner, where `A` and `B` are arrays with `double`-typed elements:

```
A = Eval1(B, std::negate<double>());
```

This would compile into a single loop like:

```
for (unsigned int i=0; i<A.Size(); i++)
{
    A[i] = f(B[i]);
}
```

Where `f` refers to the `std::negate<double>` functor object.

But is this legal? Does this functor object still exist when we finally get around to using it. The answer is yes, because the statement `A = Eval1(B, std::negate<double>());` that creates the temporary functor object, first destroys that object, when the entire statement has been executed (see section 3.1.2). The fact that the statement really compiles into a loop through the use of meta-programming, does not make any difference, as it is merely considered to be the realization or actual implementation of that statement.

Functor Object Provided, Different Argument- & Return-Types

When the functor object is provided, but the argument- and return-types are different, we may use the following version of the `Eval1`-function:

```
template <typename TRes, typename TArg, class F, class S>
Expr1<TRes, F, false, Expr<TArg, S>, false>
Eval1(Expr<TArg, S> const& expr, F const& f)
{
    return Expr1<TRes, F, false, Expr<TArg, S>, false>(expr, f);
};
```

Functor Object Not Provided

When we do not have an instance of the functor class, we must explicitly provide it as a template argument when calling the `Eval1`-function. The version of `Eval1` that supports this, is as follows:

```
template <class F, typename T, class S>
Expr1<T, F, true, Expr<T, S>, false>
Eval1(Expr<T, S> const& expr)
{
    return Expr1<T, F, true, Expr<T, S>, false>(expr);
};
```

Note that the appropriate template-argument is set to `true`, so as to indicate that the returned `Expr1`-object, must instantiate the functor in question. Since this functor must be the first template argument passed to the `Eval1`-function, an example of its usage could be:

```
A = Eval1<std::negate<double> >(B);
```

Here, the element datatype for arrays `B` and ultimately also `A`, are automatically deduced by the C++ compiler, and should of course match the datatype for the functor that is to be applied, here `double`.

Functor Object Not Provided, Different Argument- & Return-Types

When we do not have an instance of the functor class, and the argument- and return-types are different, then the following version of `Eval1` may be used:

```
template <class F, typename TRes, typename TArg, class S>
Expr1<TRes, F, true, Expr<TArg, S>, false>
Eval1(Expr<TArg, S> const& expr)
{
    return Expr1<TRes, F, true, Expr<TArg, S>, false>(expr);
};
```

Pointer To Function

When we have a pointer to a unary function, we may use the following version of the `Eval1` function:

```
template <typename T, class S>
Expr1<T, std::pointer_to_unary_function<T, T>, true,
         Expr<T, S>, false>
Eval1(Expr<T, S> const& expr, T (*f)(T))
{
    return Expr1<T, std::pointer_to_unary_function<T, T>, true,
                    Expr<T, S>, false>(expr,
                    std::pointer_to_unary_function<T, T>(f));
};
```

This may be used as follows:

```
A = Eval1(B, &std::sqrt);
```

It is important to note that we instruct the `Expr1`-class to make its own instance of the functor that it is to apply, by setting the appropriate template argument to `true`, and furthermore that its constructor takes two arguments – an expression and a functor object – and then actually copies the functor object to its own internal instance of that functor. So even though the temporary `std::pointer_to_unary_function` object that we instantiate in the code above and send to the constructor of the `Expr1`-class, is naturally destroyed when we return from that constructor; it does not matter, because the `Expr1`-object that we are returning, has actually copied that `std::pointer_to_unary_function` object to its internal functor object, and has thus stored a pointer to the supplied function `f`.

Also note that C++ does not allow us to declare `f` as being `const`. This means that we have actually provided a backdoor for having side-effects in arithmetic expressions in ArrayOps. You should not use it in such a manner though, as the code is much more error-prone and harder to understand when there are side-effects.

Pointer To Function, Different Argument- & Return-Types

When we wish to use a pointer to a function, but have different argument- and return-types, we may use the following version of the `Eval1`-function:

```
template <typename TRes, typename TArg, class S>
Expr1<TRes, std::pointer_to_unary_function<TArg, TRes>, true,
            Expr<TArg, S>, false>
Eval1(Expr<TArg, S> const& expr, TRes (*f)(TArg))
{
    return Expr1<TRes,
                 std::pointer_to_unary_function<TArg, TRes>,
                 true,
                 Expr<TArg, S>, false>(expr,
                 std::pointer_to_unary_function<TArg, TRes>(f));
};
```

3.4.2 Casting

In C++ there are four different types of casting: Static, dynamic, re-interpret, and const-casting. These casting functions are also supported for elements of arrays in ArrayOps, and are applied on an element-by-element basis.

To implement these casting functions, we really use a macro, but in the following, we will merely exemplify the principle for static-casting, where the idea is to create an ArrayOps-expression that will cast an expression using the `static_cast` function from C++. First we implement a functor that provides this functionality:

```
template <typename TRes, typename TArg>
struct static_caster : public std::unary_function<TRes, TArg>
```

```
{
    TRes operator()(TArg const& x) const
    {
        return static_cast<TRes>(x);
    }
};
```

Then we provide the function `StaticCast` for the user to call in an expression involving arrays:

```
template <typename TRes, typename TArg, class S>
Expr1<TRes, static_caster<TRes, TArg>, true,
            Expr<TArg, S>, false>
StaticCast (Expr<TArg, S> const& expr)
{
    return Expr1<TRes, static_caster<TRes, TArg>, true,
                       Expr<TArg, S>, false>(expr);
};
```

For example, let `A` and `B` be a `double`-typed array, and let `C` be of some other type (for example `int`-typed). Then the `StaticCast`-function may be used as follows:

```
A = B + StaticCast<double>(C);
```

Which effectively compiles into the following:

```
for (unsigned int i=0; i<A.Size(); i++)
{
    A[i] = B[i] + static_cast<double>(C[i]);
}
```

3.4.3 EvalAll

The `EvalAll` functions are currently only used in the assignments of ArrayOps, and they are the functions that implement the for-loops that arithmetic expressions in ArrayOps are ultimately flattened to. Different versions of the `EvalAll` functions are available, for taking right-hand arguments that are either expressions or values, and whether or not execution must be performed in parallel.

Since the boolean deciding parallelism is a template-argument in the `ArrayBase`-class, and left-hands of the `EvalAll`-functions must be such `ArrayBase`-objects, then we can switch between the parallel and non-parallel implementations at compile-time.

Non-Parallel EvalAll-Functions

The non-parallel version of the `EvalAll` that takes an expression as its right-hand argument, is as follows:

```
template <class Op, class T, class S1, class S2>
void
EvalAll(ArrayBase<T, S1, false>& arr, Expr<T, S2> const& expr)
{
    assert(MatchingSize(arr, expr));

    const Op op;
    const int kSize = (int) arr.Size();

    for (int i=0; i<kSize; i++)
    {
        op(arr[i], expr[i]);
    }
}
```

Note the `false`-valued boolean template argument for the left-hand `ArrayBase`-argument to the `EvalAll`-function, and that the right-hand argument is declared `const`, which means that there can be no side-effects. When the right-hand argument is a value instead of an ArrayOps expression, the function is as follows:

```
template <class Op, class T, class S1>
void
EvalAll(ArrayBase<T, S1, false>& arr, T const& val)
{
    const Op op;
    const int kSize = (int) arr.Size();

    for (int i=0; i<kSize; i++)
    {
        op(arr[i], val);
    }
}
```

There should be no surprises here, although you may note that we instantiate the functor object for the operator outside the for-loops, because this could potentially be expensive for non-trivial functors. The size is converted to a signed integer, so as to have an implementation that is almost identical to the parallel version below.

Parallel EvalAll-Functions

The parallel versions of the `EvalAll` functions use OpenMP [Board] to implement parallelism in the for-loop. Typically the operator functor contains no data at all, and merely provides raw functionality (such as assignment, or accumulative assignment). At any rate, the functor object for the operator is declared `const`, to disallow side-effects. When the right-hand argument is an expression, the parallel `EvalAll` function is as follows:

```
template <class Op, class T, class S1, class S2>
void
EvalAll(ArrayBase<T, S1, true>& arr, Expr<T, S2> const& expr)
{
    assert(MatchingSize(arr, expr));

    const Op op;
    const int kSize = (int) arr.Size();

    #pragma omp parallel for if (kSize>=arr.GetParLimit())
    for (int i=0; i<kSize; i++)
    {
        op(arr[i], expr[i]);
    }
}
```

When the right-hand argument is a value, the parallel `EvalAll` function is:

```
template <class Op, class T, class S1>
void
EvalAll(ArrayBase<T, S1, true>& arr, T const& val)
{
    const Op op;
    const int kSize = (int) arr.Size();

    #pragma omp parallel for if (kSize>=arr.GetParLimit())
    for (int i=0; i<kSize; i++)
    {
        op(arr[i], val);
    }
}
```

OpenMP requires for the size to be a signed integer, although the ArrayOps framework uses unsigned integers. This is the reason why we cast the array-sizes in all of these functions, to keep them as identical as possible.

3.4.4 ReduceAll

In ArrayOps the `ReduceAll` set of functions provide basic functionality for reducing an array into a single scalar value. The usage of these functions was detailed in section 2.6.6 on page 28, and here we shall only recap the fact that we use an accumulative functor that is assumed to provide two operator-overloadings: One unary, and one nullary. The unary overloading of `operator()` takes an argument of appropriate type and accumulates it to some internal variable. The overloading of `operator()` that takes zero arguments, is a simple query of the result, which may or may not perform some final calculations.

The different versions of the `ReduceAll` functions are similar in flavour to the `Eval1` set of functions from page 60, in that they provide the user with the opportunity to call the `ReduceAll` function with or without a functor object, and with either matching or differing argument- and return-types.

Functor Object Provided

When the accumulative functor object is provided, the following version of the `ReduceAll`-function should be used:

```
template <class F, class T, class S>
T ReduceAll(Expr<T, S> const& x, F& f)
{
    assert(x.IsSized());
    const unsigned int kSize = x.Size();

    for (unsigned int i=0; i<kSize; i++)
    {
        f(x[i]);
    }

    return f();
}
```

Note that the ArrayOps expression which is reduced, is declared `const`, and hence does not change. This is due to the fact that ArrayOps should not allow side-effects in arithmetic expressions, and we may of course pass any kind of arithmetic expression to the `ReduceAll` function. But the functor object computing the actual reduction, can of course not be declared `const`.

Functor Object Provided, Different Argument- & Return-Types

When the accumulative functor object is provided, and the argument- and return-types are different, the following version of the `ReduceAll`-function should be used:

```
template <class TRes, class TArg, class F, class S>
TRes ReduceAll(Expr<TArg, S> const& x, F& f)
{
    assert(x.IsSized());
    const unsigned int kSize = x.Size();

    for (unsigned int i=0; i<kSize; i++)
    {
        f(x[i]);
    }
```

```
    return f();
}
```

Functor Object Not Provided

When the accumulative functor object is not provided, the following version of the `ReduceAll`-function should be used:

```
template <class F, class T, class S>
T ReduceAll(Expr<T, S> const& x)
{
    assert(x.IsSized());
    const unsigned int kSize = x.Size();

    F f;

    for (unsigned int i=0; i<kSize; i++)
    {
        f(x[i]);
    }

    return f();
}
```

Functor Object Not Provided, Different Argument- & Return-Types

When the accumulative functor object is not provided, and the argument- and return-types are different, the following version of the `ReduceAll`-function should be used:

```
template <class F, class TRes, class TArg, class S>
TRes ReduceAll(Expr<TArg, S> const& x)
{
    assert(x.IsSized());
    const unsigned int kSize = x.Size();

    F f;

    for (unsigned int i=0; i<kSize; i++)
    {
        f(x[i]);
    }

    return f();
}
```

3.5 Reductions

As described on page 31, there are essentially two kinds of reductions in ArrayOps: Reductions that result in a single scalar value of the same type as the array that is reduced, and reductions that result in a single scalar value of some other type. We saw an example of the former on page 31, which was the `Sum`-function that computes the summation of an array's elements.

3.5.1 Reduce-Class

Some kinds of reductions may result in fractional numbers. This is for example the case with the `Mean`-function for computing the average of the array's elements. Most often we will probably just use the built-in datatypes from C++, and would hence like this average returned as a `double`-typed value. However, there may be cases where you want the internal calculations of a reduction, to use a custom datatype with higher precision, and hence will need to instruct the function computing the reduction about this. But since C++ does not allow default values for the template arguments of functions (C++ only allows this for classes), then we must split the implementation of these kinds of reductions in two: One part that can return arbitrary types, and one part which merely uses the `double`-type, for convenience. Let us start with the former, which we wrap in a template class:[10]

```
template <typename TRes=double>
class Reduce
{
public:
    // Mean
    template <typename TArg, class S>
    static inline
    TRes Mean(Expr<TArg,S> const& x)
    {
        assert(x.IsSized());
        assert(x.Size()>0);

        return ((TRes) Sum(x))/x.Size();
    };
};
```

Since the functions in this class are declared `static`, we do not have to instantiate an object before we can use those functions. So it can be used directly, in a manner like this:

```
double avg = Reduce::Mean(A);
```

[10] This class actually holds other functions for computing reductions as well, such as the `Norm` and `Variance` functions.

Where `A` is some array. This notation however, does not seem all that convenient, so ArrayOps provides another function as well, which simply wraps this for us, for the case where the return-value is `double`-typed:

```
template <typename T, class S> inline
double Mean(Expr<T,S> const& x)
{
    return Reduce<double>::Mean(x);
};
```

Which allows us to write the more comprehensible:

```
double avg = Mean(A);
```

3.6 Arrays

In the ArrayOps framework, arrays are themselves considered expressions, and must therefore implement the `Expr`-class described in section 3.2.4. We shall not give the implementations of all the different array-classes here, but suffice with their base-class `ArrayBase` and a few good examples of arrays; the fixed-size `ArrayMini`-class, and the automatically resizable `ArrayAuto`-class which derives from the `Array`-class. Hopefully you will understand the implementation from this, and be able to write array-classes of your own, should the default ones not meet your needs.

3.6.1 Assignment

Before we describe the actual array implementations, let us first look at assignment between objects in C++. First off, in C++ we must implement the assignment operators in the class itself, so even though we have a common base-class for all arrays (the `ArrayBase`-class), we are not able to just implement assignment there. However, this is not the case for accumulative assignment operators (such as `+=`, `-=`, and so on), which can indeed be implement for just the common base-class.

Beyond the accumulative assignment overloadings, there are three kinds of assignments that we must be able to handle in any array-class in ArrayOps:

- Assignment from an array-object of the exact same class. (More on this below.)
- Assignment from any ArrayOps expression.
- Assignment from a value. That is, each element in the array must be assigned the same value.

Typically, you will just use a macro for making all the assignment functions in your array-class. An example of such a macro is given below.

Assignment From Exact Same Classes

In C++, performing assignment on objects of the exact same class, defaults to assignment for each of the objects' member-fields. For classes with many member-fields, this is often a nice feature of the language, but in some cases it could give rise to disastrous bugs if we do not override this kind of assignment properly.

Take for example an array that allocates storage and saves a pointer to this storage. If we had not provided a specialized assignment operator for this kind of array, such an assignment would result in just the pointer getting assigned, instead of actually copying the elements of the arrays, as we intend – and this is even though we might have implemented the assignment operator which takes a right-hand that is an `Expr`-object, which the array-class itself inherits from. Assignment from the exact same class still defaults to assignment of all of the individual member-fields, and this assignment takes precedence as it is a better match.

Exact Same Template Class?

The classes that are used to implement arrays in ArrayOps, are all template classes. So implementing assignment from the exact same class, naturally raises the question: What is the exact same class of a template class? Is it any choice of template arguments, or must the template arguments be the exact same as well, for the classes to be considered identical?

The answer is that the template arguments must be identical. If they are not, then the assignment will be handled by the case that assigns from an `Expr`-object. Fortunately, this makes things a bit simpler, in that a template class may be referred inside the class itself, without writing the template arguments, thus assuming the template arguments are the same as provided during the construction of the class. An example of this is found in the description of the `ArrayMini`-class in section 3.6.3 below.

Assignment From Self

Often when overriding the assignment operator for some class in C++, one also checks to see if assignment is performed on the object itself (that is, an assignment such as `A = A;`), this however, is not really a bug in ArrayOps, although since side-effects are disallowed, the assignment would be superfluous. However, if we were to add an `if`-statement skipping such assignments, then we would add extra overhead, accounting for a situation that is likely never to occur, and even if it did, it would not pose any danger at all. For this reason, such checking has been omitted.

Assignment Macro

To create all the different assignments inside a class, a macro called `AOp_MakeAssign` is being used. Let us use the `ArrayMini`-class as an example, and see the output

of the macro for regular assignment:

```
// Assignment from exact same class.
inline
ArrayMini& operator= (ArrayMini const& arr)
{
    EvalAll<assign<T> >(*this, arr);

    return *this;
}

// Assignment from arbitrary Expr-objects.
template <class S1> inline
ArrayMini& operator= (Expr<T, S1> const& expr)
{
    EvalAll<assign<T> >(*this, expr);

    return *this;
}

// Assignment from value.
inline
ArrayMini& operator= (T const& val)
{
    EvalAll<assign<T> >(*this, val);

    return *this;
}
```

Note how the assignment functions are just implemented by applying the `EvalAll`-function with an appropriate functor, in this case called `assign`, which simply applies `operator=` on each of the elements in the arrays.

For accumulating assignment operators (such as `+=` or `*=`), the macro creates functions similar to the above, but leaves out assignment from the exact same class, as this does not default to anything for that kind of assignment in C++, and therefore does not need special attention.

3.6.2 ArrayBase-Class

All arrays must derive from the `ArrayBase`-class, and must provide a sized array whose elements can be looked up, as well as having values assigned to them.

Class Layout & Constructors

The `ArrayBase`-class is of course a template class, taking the usual datatype `T` for its elements, the implementor-class `S`, and finally a boolean template argument `Parallel`, deciding whether or not to execute assignments in parallel,

whenever an instance of the class occurs as a left-hand in an assignment. The basic class layout is as follows:

```
template <class T, class S, bool Parallel>
class ArrayBase : public Expr<T, S>
{
public:
    ArrayBase (unsigned int parLimit=kParLimit) :
        Expr<T, S>(), mParLimit(parLimit) {}

    ArrayBase (S const& s,
               unsigned int parLimit=kParLimit) :
        Expr<T, S>(s), mParLimit(parLimit) {}

    // ...
};
```

Note that the second of these constructors, passes an instance of the implementor-class `S` to the super-class `Expr`, which in turn passes it to its super-class `S`.

The parallel limit `kParLimit` is set to 1000 elsewhere, which may change in future versions of ArrayOps. The rest of the member functions of the `ArrayBase`-class, are detailed below.

Parallelism Activation Limits

The `ArrayBase`-class also provides functions for querying and altering the parallelism limits:

```
// Get/Set the limits for when to use parallelism.
inline int
GetParLimit () const { return mParLimit; }

inline void
SetParLimit (int parLimit) { mParLimit = parLimit; }
```

And the integer variable `mParLimit` is declared `private` inside the `ArrayBase`-class.

Checked Lookup Function

The `ArrayBase`-class provides the non-`const` version of the checked lookup function, that throws an exception if the index is out of range. The function merely uses the `operator[]` function that is assumed to be provided by the implementor-class supplied through the template argument `S`, and is hence implemented as follows:

```
// Element lookup, non-const.
T& at(unsigned int i)
```

```
{
    if (i >= S::Size())
    {
        throw std::out_of_range(kErrRange);
    }

    return (*this)[i];
}
```

The `const`-version of this function was declared in the `Expr`-class (see page 51).

Index Manipulator Creators

The slice, cycle, and reverse index manipulators also reside as members of the `ArrayBase`-class, whilst themselves deriving from the `ArrayBase`-class, thus allowing for assignments and such to be performed on them as well. To implement this in C++ we must use forward declarations. To make instantiation of these classes for index manipulation easier, functions have been provided that set the `ArrayBase`-object on which to apply the index manipulator, to be the `this`-object. The implementation inside the `ArrayBase`-class is as follows:

```
// Forward declarations for different
// index-manipulation arrays.
class ArraySlice;
class ArrayCycle;
class ArrayReverse;

// Return an ArraySlice-object.
inline ArraySlice
Slice (unsigned int offset, unsigned int size)
{
    return ArraySlice(*this, offset, size);
}

// Return an ArrayCycle-object.
inline ArrayCycle
Cycle (unsigned int offset=0)
{
    return ArrayCycle(*this, offset);
}

// Return an ArrayReverse-object.
inline ArrayReverse
Reverse ()
{
    return ArrayReverse(*this);
}
```

And the actual implementations for the classes `ArraySlice` and so on, are then defined elsewhere (see section 3.7 below).

Fail-Safe Functions

The following functions are declared `private` to ensure that no initialization assignments take place, and that no unhandled assignments take place, in case you had forgotten to implement the `operator=` for an array-class you made:

```
// Ensure no implicit assignments take place.
ArrayBase (ArrayBase const& arr) { assert(false); }

// Ensure no unhandled assignment takes place.
template <class X>
ArrayBase& operator= (X const& x) { assert(false); }

// Ensure no unhandled assignment takes place.
template <class X>
ArrayBase& operator= (X& x) { assert(false); }
```

This should not be able to compile at all, so the assert-statements are just there to show you that we should absolutely not get to that point.

Let us say you have made an array-class called `MyArray`. The trouble with having left out the `operator=` overloading in the `MyArray`-class, for when the right-hand argument is a `MyArray`-object as well, is that C++ then defaults to copying each of the member-fields in the `MyArray` instance, as described on page 71. If this includes, say, pointers to memory that has been allocated, then it is clearly a bug. So once more, it was chosen that ArrayOps should help you write proper code and avoid bugs.

3.6.3 ArrayMini-Class

The following sections describe the `ArrayMini`-class, which implements an array whose size is fixed and known at compile-time. This means the compiler may be able to optimize the code better, and the array-type is actually intended to be used with smaller arrays, whose operations may then be entirely unrolled and inlined by the compiler.

Implementor

The `ArrayMini` implementor that contains the actual storage and the inlined access functions, is as follows:

```
template <class T, unsigned int kSize>
class ArrayMini_Imp
{
public:
    ArrayMini_Imp () {}
```

```
    // Element lookup.
    inline T& operator[] (unsigned int i)
    {
        assert(i<kSize);
        return mStorage[i];
    }

    // Element lookup, const.
    inline
    T const& operator[] (unsigned int i) const
    {
        assert(i<kSize);
        return mStorage[i];
    }

    // Size of array.
    inline unsigned int Size () const { return kSize; }
    inline bool IsSized () const { return true; }

protected:
    T mStorage[kSize];
};
```

Instantiator

The `ArrayMini`-class that you should instantiate, and which basically just implements the assignment operators as well as deriving from the `ArrayBase`-class with the `ArrayMini_Imp`-class as its implementor, is given as follows:

```
template <class T, unsigned int kSize, bool Parallel=false>
class ArrayMini :
        public ArrayBase<T, ArrayMini_Imp<T, kSize>, Parallel>
{
public:
    ArrayMini () :
        ArrayBase<T, ArrayMini_Imp<T, kSize>, Parallel>() {}

    // Macro making overloadings for the assignment operators.
    AOp_MakeAssign(ArrayMini);
};
```

Naturally, assignment between two objects of the exact same `ArrayMini`-class (that is, two instances of the `ArrayMini`-class, whose template arguments are identical), would be handled properly by the default assignment as described in section 3.6.1, as the only memberfield the class has, is the `mStorage` array of elements, which would get copied element-by-element, according to C++

semantics. However, this would of course ignore whether or not to execute the assignment in parallel through the use of the `EvalAll` function, and so, we must implement assignment from `ArrayMini`-objects as well, as shown in the source-code above, by the use of the generic `AOp_MakeAssign`-macro.

3.6.4 ArrayAuto-Class

The following describes the `ArrayAuto`-class, which implements an array whose size is automatically adjusted in an assignment, whenever such an object occurs in the left-hand of an assignment expression, and where the right-hand is of a different size. This array-class is an example on how to extend on the functionality of an existing array-class, by way of ordinary C++ class-inheritance.

Instantiator

The `ArrayAuto`-class has only an instantiator, deriving from the `Array`-class, which is assumed to implement a function called `Resize` and a function called `ResizeCopy`, as described in section 2.3.2 on page 10. The `ArrayAuto`-class is implemented as follows:

```
template <class T, bool Parallel=true>
class ArrayAuto : public Array<T, Parallel>
{
public:
    ArrayAuto () : Array<T, Parallel>() {}
    ArrayAuto (unsigned int size) : Array<T, Parallel>(size) {}

    ArrayAuto (ArrayAuto const& x) : Array<T, Parallel>()
    {
        *this = x;
    }

    template <class S1>
    ArrayAuto (Expr<T, S1> const& x) : Array<T, Parallel>()
    {
        *this = x;
    }

    // Macro making overloadings for the assignment operators.
    // This macro is specialized for the ArrayAuto-class.
    AOp_MakeAssignAuto(ArrayAuto);
};
```

Assignment Macro

The assignment overloadings generated by the `AOp_MakeAssignAuto`-macro, are quite similar to those on page 71 for the `AOp_MakeAssign`-macro, only we now

have to check if the size of the array must be adjusted. Let us first see how assignment from the exact same class is implemented:

```
// Assignment from exact same class.
inline
ArrayAuto& operator= (ArrayAuto const& x)
{
    assert(x.IsSized());

    if (Size() != x.Size())
    {
        Resize(x.Size());
    }

    EvalAll<assign<T> >(*this, x);

    return *this;
}
```

Note how we call the function `Resize` if the sizes do not match. Recall that the `Resize` function does not copy the old elements of the array. An example of an accumulative assignment operator that uses the `ResizeCopy` function instead, is implemented as follows:

```
// Assignment from arbitrary Expr-objects.
template <class S1> inline
ArrayMini& operator+= (Expr<T, S1> const& x)
{
    assert(x.IsSized());

    if (Size() != x.Size())
    {
        ResizeCopy(x.Size());
    }

    EvalAll<assign_plus<T> >(*this, x);

    return *this;
}
```

Note however, that in case the new array-size is greater than previously, then the new array-elements are not initialized, so that part of the array will contain garbage. Also note that resizing of the array is not necessary when the right-hand is a value, so that kind of operator is implemented as it was for the `AOp_MakeAssign` macro.

3.7 Index Manipulators

The index manipulators, that is, the slice, cycle, and reverse arrays, are merely wrappers that map the index of another array in some fashion. This means we should merely hold a reference to the array whose index is to be mapped, and then store the mapping attributes, which will then be used in the lookup functions of the wrapper class. In the following we will describe the implementation of the `ArraySlice` class, which is a good example of this. The other index manipulation classes are implemented similarly.

3.7.1 Slice

An array-slice is a reference to another array, whose index is offset and whose size may be smaller than that of the original array.

Implementor

The implementor-class containing storage and the inlined lookup and size functions, is implemented as follows:

```
template <class T, class S, bool Parallel>
class ArraySlice_Imp
{
public:
    ArraySlice_Imp (ArrayBase<T,S,Parallel>& arr,
                    unsigned int offset,
                    unsigned int size) : mArray(arr),
                                         kOffset(offset),
                                         kSize(size) {}

    // Element lookup.
    inline T& operator[] (unsigned int i)
    {
        assert(i<kSize);

        return mArray[i+kOffset];
    }

    // Element lookup, const.
    inline
    T const& operator[] (unsigned int i) const
    {
        assert(i<kSize);

        return mArray[i+kOffset];
    }
```

```
    // Size of array.
    inline unsigned int Size () const { return kSize; }
    inline bool IsSized () const { return true; }

protected:
    ArrayBase<T,S,Parallel>&  mArray;
    const unsigned int        kOffset;
    const unsigned int        kSize;
};
```

Note the use of assertions in the lookup functions, where we check that the index is within proper bounds, even though the referenced `ArrayBase` object presumably also does this. This excessive use of assertions, makes debugging much easier, as the origin of a bug is better pinpointed. It also makes it easier to read from the source-code, what parameters are considered valid.

Also note the attributes being declared `const`, which may assist the compiler in making optimizations of the code.

Instantiator

The `ArraySlice`-class that is instantiated through the `Slice()` function in the `ArrayBase`-class, is implemented as follows:

```
template <class T, class S, bool Parallel>
class ArrayBase<T,S,Parallel>::ArraySlice :
    public ArrayBase<T, ArraySlice_Imp<T,S,Parallel>, Parallel>
{
public:
    // Convenient type-definition of implementor.
    typedef ArraySlice_Imp<T, S, Parallel> TImp;

    ArraySlice(ArrayBase& arr,
               unsigned int offset,
               unsigned int size) :
        ArrayBase<T, TImp,
                  Parallel>(TImp(arr, offset, size)) {}

    // Macro making overloadings for the assignment operators.
    AOp_MakeAssign(ArraySlice);
};
```

There are no surprises here, as the implementation follows that of any other array in ArrayOps, such as `ArrayMini` above, which means that an `ArraySlice`-object may also be used as an lvalue in arithmetic expression.

3.8 Object Destruction

Usually when one has a class-hierarchy in C++, and the sub-classes must have their destructors called in all situations, regardless of whether we know it to be a certain sub-class or not, one declares the destructor of the super-class to be `virtual`. We could for example have a super-class `A` as follows:

```
class A
{
public:
    virtual ~A() {}

    // ...
};
```

And a sub-class `B` could then be:

```
class B : public A
{
public:
    ~B() { /* Specialized cleanup */ }

    // ...
};
```

Now, if we were to have some function `Foo` taking as argument a pointer to some instance of the class `A`, and if the function ultimately deletes this instance of `A`, as follows:

```
void Foo(A* a)
{
    // ...

    delete a;
}
```

then even though there is no way of knowing if the object `a` is really an instance of just the class `A`, or if it is actually an instance of its sub-class `B`, the destructor of `B` will be called whenever the latter is the case, because the destructor in class `A` was declared `virtual`. If it had not been declared `virtual`, then the destructor of class `B` would not be called.

3.8.1 Code Generation

Unfortunately, the C++ compiler generates some extra code for the construction of an object, whenever a class has one or more virtual functions. So even though the destructor is only called once in the lifetime of an object, we would like to avoid the class having `virtual` functions altogether, including the destructor.

Fortunately, the classes in the ArrayOps framework do not need any specialized cleanup code, except for some of the classes implementing the actual arrays. So the direct approach would be to simply implement the destructors in the classes that need specialized cleanup. However, this would mean that we always had to know exactly which class an object was an instance of, in order for the destructor to be invoked.

3.8.2 Implementor Class & Cleanup Code

The solution is therefore, to always ensure that the cleanup code is in the socalled implementor. That is, the cleanup code must be in the class that serves as the super-class to the `Expr`-class from section 3.2.4, which then allows us to have a function like the following:

```
template <class T, class S>
void Goo(Expr<T, S>* x)
{
    // ...

    delete x;
}
```

Which would just call the destructor of the class `S`, as the class `Expr<T, S>` does not have a destructor by itself.

3.8.3 Array-Class

An example of this is found in the `Array`-class, whose implementor-class has both a constructor and destructor:

```
template <class T>
class Array_Imp
{
public:
    Array_Imp  () : mStorage(0), mSize(0) {}
    ~Array_Imp () { DoDelete(); }

    // ...
};
```

Where allocation and deletion is actually implemented in the functions named `DoAllocate` and `DoDelete`. Note that the constructor of `Array_Imp` does not allocate any storage, as this is assumed to be done by the instantiator-class as we shall see next:

```
template <class T, bool Parallel=true>
class Array : public ArrayBase<T, Array_Imp<T>, Parallel>
{
```

```
public:
    Array () : ArrayBase<T, Array_Imp<T>, Parallel>() { }
    Array (unsigned int size) :
        ArrayBase<T, Array_Imp<T>, Parallel>()
    {
        DoAllocate(size);
    }

    // ...
};
```

The `Array`-class is the class that one actually instantiates, and as can be seen, it derives from the `ArrayBase`-class, which in turn derives from the `Expr`-class, which ultimately derives from the class `Array_Imp` through reverse inheritance. This entire class-hierarchy should only have a single destructor implemented, and it should always be in the top-most super-class, here the class `Array_Imp`, which is also the case.

Bibliography

OpenMP Architecture Review Board. Openmp. URL `http://www.openmp.org/`.

MPI committee. Message passing interface (mpi). URL `http://www-unix.mcs.anl.gov/mpi/`.

Free Software Foundation. Gnu lesser general public license. URL `http://www.gnu.org/copyleft/lesser.html`.

Steve Karmesin, Scott Haney, Bill Humphrey, Julian Cummings, Tim Williams, Jim Crotinger, Stephen Smith, and Eugene Gavrilov. Parallel object-oriented methods and applications (pooma). URL `http://acts.nersc.gov/pooma/`.

Bjarne Stroustrup. *The C++ Programming Language, Second Edition*. Addison-Wesley, 1991. ISBN 0-201-53992-6.

Edwin Robert Tisdale. The scalar, vector, matrix and tensor library (svmtl). URL `http://www.netwood.net/~edwin/svmtl/`.

Todd Veldhuizen. Blitz++. URL `http://www.oonumerics.org/blitz/`.

Index