Expresso
2.0.2

Generated by Doxygen 1.5.6

Mon Feb 23 19:15:45 2009

Contents

Chapter 1

Main Page

This document describes the C++ array class template package **Expresso** which provides efficient and flexible arrays for numerical computations with arbitrary array element types.

The library is available for download under the LGPL copyleft terms (see `COPYING`) from the `Download page` Printable versions of this document (pdf and ps) are available on the above download page.

1.1 Key features

- **Easy** As simple and straightforward to use as the c++ language syntax allows.
- **Safe** memory handling is completely automatic using reference counting. No memory leaks.
- **Efficient** by using expression template techniques there is no need for temporary objects in expressions. Fast elementwise access with parenthesis operator.
- **Flexible:** The rank and the element type is arbitrary (template arguments). The size and index ranges of an array can be reallocated at runtime. Dimensions may be permuted and reversed, much like in matlab. Arrays may refer to subsets of other arrays.
- **Fortran compatible** Pointers can be passed to fortran routines where they are declared as fortran arrays and indexed as usual. The indexing convention is Fortran (innermost rank first).
- **Extendible** The expert may define new array specializations with new ways of storing data, for example banded, sparse ... If certain restrictions are fulfilled, these array types may be used with any other array types or scalars to form expressions using the template operators.

1.2 Platforms

Everything is strict ANSI standard C++. Most compilers now implement all the features of ANSI C++ that are needed by expresso.

The following compilers have been found to work fine with expresso. Newer versions of the same compiler is very likely to work just as well.

- gcc 3.0

- intel icc 7.0
- Digital Dec cxx 6.0
- Portland group C++ 1.7
- KAI C++ 3.2f
- MIPSpro Compilers: Version 7.41

The following old compiler versions have been found to fail compiling expresso:

- gcc 2.8.1
- Sun C++ 4.2
- Microsoft Visual C++ 6
- MIPSpro Compilers: Version 7.20

1.3 Quick start

Use the expresso array template as follows

- Unpack the package where you want it

```
> cd /usr/local/include
/usr/local/include> gunzip < expresso.tgz | tar xf -
```

- Edit your sourcefile in the working directory for example as follows:

```
#include "expresso.h"
using namespace esso;

main() {
  Array<3,float> x(10,20,30);

  // You have now a fortran compatible 3-dimensional array of floats
  // with index range (0:9,0:19,0:29)

  x<<=0;            // Efficient elementwise copy from scalar
  x(2,5,14)=1;      // Fortran-like element access
  x<<=25*Sqrt(x)-x; // Efficient arithmetic expressions
}
```

- Compile your program with an ANSI-C++ compiler. The performance depends on the compilers ability to optimize the code, expresso gives the compiler maximal opportunities to do a good job. The free compiler gcc will give fast code with the following options:

```
> gcc -I/usr/local/include/expresso -O3 testc.cc
```

No special libraries need to be linked in.

1.4 An expresso tour

This section contains a more thourough introduction to expresso. The most important features are presented. After reading this short text, you should be able to use expresso for most tasks, however, first read the exremely short "Quick start" section.

1.4.1 The Array template

An **esso::Array** (p. 41) object is declared as follows:

```
Array<3,float> a(10,20,30);
```

where '3' is the **rank** (i.e. the number of dimensions), 'float' is the element type and the parameters 10,20,30 are index ranges. The index range of this array is (0:9, 0:19, 0:29). The corresponding fortran declaration would be

```
real a(0:9,0:19,0:29);
```

There is a second flavor of Array which may give more efficient code but is somewhat less flexible.

```
Array<3,Fix<float> > a(10,20,30);
```

The difference is a choice between speed and flexibility: the standard Array is more flexible and Fix is faster.

The most useful constructors for Arrays follow (other exist), see **esso::Array** (p. 41) class documentation):

```
Ix<3> ix(10,20,30);

// Range as defined in 'ix'
Array<3,float> a(ix);

// An array with start index is 1 instead of 0
Array<3,float> b(ix,Ix<3>(1));

// Copy constructor, this creates a reference to the same elements
Array<3,float> c(b);
```

The class template `Ix` is a small vector of integers whose length is known at compile time, see **esso::Ix** (p. 58).

1.4.2 Element access

A simple, powerful and fast way of accessing an element is using operator (). This works just like in fortran:

```
Array<3,Fix<float> > x(10,10,10);

for(int k=x.L(2); k<x.U(2); k++)
  for(int j=x.L(1); j<x.U(1); j++)
    for(int i=x.L(0); i<x.U(0); i++)
      x(i,j,k)=x(i,j,k)*x(i,j,k)-1;
```

This produces optimal code in many cases, because the index operator is inlined and carefully designed for optimal performance. (The Fix Array specialization is recommended for best performance). However, in general array expression techniques (see the section on Arithmetic operations) give equal or better performance and in addition more safety because the programmer need not be concerned with index bounds.

Note the usage of member functions L and U. These functions return lower and upper index bounds for the Array.

Just as in fortran, the innermost loop should be over the first index for best performance, this is because the elements are by default stored in memory in the same way as fortran (unless you have permuted the ranks in your code). Note that the storage order of ranks is thus by default reversed compared to built-in C and C++ arrays.

1.4.3 The Ix template

The **esso::Ix** (p. 58) template is used to specify an small vector. An Ix object can be declared as follows:

```
Ix<3,T> i;
```

Replace T by the desired element type (default is a signed integer type) The index elements are accessed with operator []. The obvious constructors and operators are defined:

```
Ix<3> i(10,15,20);   // All elements are initialized
Ix<3> j(10);         // 10 is used to initialize all elements
Ix<3> k;             // Uninitialized
Ix<3> l(i);          // Elements of Ix object i are copied

j[1]=25;             // Element access
j=25;                // Assign all elements with 25
j=i;                 // Copy elements from Ix object
cout << j[0] <<' '<< j[1] <<' '<< j[2] <<endl;
```

The output of the above code is "10 25 20".

Index objects can (of course) be used to access elements in an array:

```
Ix<2> i(2,2);

Array<2,float> x(i);   // Allocate array using length given by i

i = Ix<2>(1,0);
a[i] = 3.15;      // Access an element using an Ix object as index
cout << a;
```

Note that the stream operator is defined for Array objects.

It is recommended that the ()-operator are used for indexing an Array rather than Ix objects when performance is important, because most of todays compilers are better at optimizing such code. Ix objects are still very commonly used though, for example as arguments to logical operators and to declarations of Arrays.

Another possibility is to use Ix objects as element types for Array objects. The following code snippet shows some of the flexibility by such technique:

```
Array<2,Ix<3,float> > a(10,20);
a(2,4)[1] = 0;          // Access a single float element
a(2,4) = 1;             // Access a Ix object and assign with a scalar
a <<= Ix<3,float>(1,2,3); // Assign all Ix elements with an Ix object
a <<= 3.14;             // Assign all float elements with a scalar
```

1.4.4 Reference counting

One or more Array objects can refer to the same memory block, or different parts of the same memory block. This is all handled automatically, so the programmer need seldomly be concerned with the memory allocation.

How is this done?

When an array is created memory is allocated automatically to fit it's declared index range (unless declared with the default constructor, in which case no memory is allocated). When another array is assigned from the first, a counter in the memory block object is increased, so that at all times the memory block knows how many Array objects are referring to it. When an Array object stops referring to a certain memory block the counter is decreased one step. When the counter reaches zero, the memory block is deallocated automatically.

As an example, it is safe and reasonably efficient to return a locally declared Array object from a function, because the memory will continue to exist until the last Array object referring to it is destroyed or assigned some other memory block.

```
Array<3,float> f(float q) {
  Array<3,float> array(10,10,10);
  array<<=q;
  return array;       // Safe and fast!
}

main() {
  Array<3,float> x;

  f(2.8);             // The returned memory is safely deleted

  x=f(3.14);          // x now points to the memory allocated in the
                      // call to f()
}
```

Other examples where arrays refer to the same memory area:

```
Array<3,float> a(10,10,10);

Array<3,float> b(a);   // b refers to the same memory as a
Array<2,float> c;
c = Rmdim(a,2,5);            // c refers to the slice resulting
                             // from fixing dimension 2 to index value 5

void f(Array<3,float>);

f(a);                        // The local array object in f
                             // refers to the same memory as a
```

What is a slice? It is the (N-K)-dimensional array obtained by taking a N-dimensional array and specifying K of it's indexes.

In many senses the Array objects work as a pointer with automatic memory allocation mechanism and an index range.

1.4.5 Assignment operations

If elementwise assignment is wanted, use operator <<=

```
Array<3,float> a(10,10,10),b(15,10,5);

a<<= 3.14;                   // Copy 3.14 to all elements

b<<=a;                       // Copy elements from Array a to b
```

If the index ranges differ between the source and destination in an assignment (as in the above example) the loops are only performed over the intersection of the source and destination array index ranges. For more info on assignment and expressions of arrays, see documentation of operator <<=.

The operators +=, -=, *= and /= work the same way as <<= but also performs an arithmetic operation.

To create a reference rather than copying elements, use operator =

```
Array<3,float> a(10,10,10),b(15,10,5);

a<<= 3.14;                  // Copy 3.14 to all elements

b=a;                        // Do not copy elements, create reference
```

In the above example, the index range of `b` is identical to that of `a` after the assignment.

1.4.6 Logical operations

An array can be manipulated in many more ways at runtime than is possible in fortran 77 or fortran 90. The index range can be shifted or restricted, dimensions may be permuted, and the memory block can be reinitialized with a new memory area. These manipulations (which do not change the actual values of the elements) are refered to as Logical array operations.

```
Array<3,float> a(10,10,10);

a.Shift(0,10);          // The index range now is (10:19,0:9,0:9)
a.Permute(0,2);         // Ranks 0 and 2 are permuted
a.Reverse(1);           // The order of elements in dimension 1 is reversed
```

Many more logical functions exist that takes different arguments, see **esso::Array** (p. 41) class documentation.

These member functions also exist in a function flavors:

```
Array<3,float> a(10,10,10),b,c,d;
b = Shift(a,0,10);
c = Permute(a,0,2);
d = Reverse(a,1);
```

See **esso** (p. 15) namespace documentation for listing of more logical functions.

1.4.7 The Permutation class

To improve safety and convenience, a special class is provide to help dealing with permutations of ranks. Using this object you can create elementary permutations easily, permute arrays and Ix objects using any permutation, compose permutations as product of other permutations, invert and transform permutations in different ways. See **esso::Permutation** (p. 63) for more details.

1.4.8 Arithmetic operations

The usual arithmetic operations can be used to form expressions that can in turn be used as right hand side in an array assignment operation. Any combination of arrays and scalar (element types) can be mixed in expressions. (The arrays must have the same rank of course).

```
Array<2,float> a,b,c,d;
a<<=a-b*c+3.14;
```

The assignment is done over the intersection of all arrays.

In addition to the four arithmetic operations there are Max, Min, Abs, Sqr, Sqrt, Pow and some other functions defined to be used in arithmetic expressions, see **esso** (p. 15) namespace documentation for a complete listing.

Of course, the result of logical functions can also be used in arithmetic expressions. (However, arithmetic expressions can in most cases not be used as input to logical functions).

When compiled, the assignment with an arithmetic expression right hand side results in a single loop, i.e. no temporary array objects are created. This means that a good compiler may produce optimal code for such array expression assignments.

Experience with some of todays compilers indicate that best performance is attained by forming expressions that do not contain to many operands. As a rule of thumb, use of ten or more array operands is not recommended. Also, using arithmetic assignment operators (i.e. +=, -= ...) wherever possible is recommended. (i.e. rather than a<<=a+1, use a+=1).

1.4.9 Fixed index range

The Fix type arrays can have one or more of it's innermost index ranges fixed at compile time. The syntax is as follows:

```
Array<4,Fix<float[30][20]> > a(10,5); // Index range is (0:29,0:19,0:9,0:4)
a(0,1,10,20) = 3.14;    // The fixed ranks are indexed just as usual.
```

All index ranges can not be fixed, at least one rank must be variable (namely the last).

Note that an alternative is to use Ix objects as elements:

```
Array<2,Ix<20,Ix<30,float> > > a(10,5); // Index range is (0:29,0:19,0:9,0:4)
a(0,1)[10][20] = 3.14;
```

There are advantages and disadvantages with both techniques, choose the way you find most convenient and efficient for your application.

1.4.10 MPI support

For parallel programs using MPI expresso provides functions for creating an MPI datatype corresponding to an expresso array. Both standard C MPI bindings and C++ bindings are supported. See **esso::CreateMPIArray** (p. 21) and **esso::CreateMPIPPArray** (p. 21) You must include expresso/mpi.h and define USE_MPI or USE_MPIPP macro to use these functions.

1.4.11 Debugging support

Defining the macro EXPRESSO_DEBUG activates a large number of runtime tests in the expresso code. For instance, all array access operations are checked against the valid index bounds. Almost all arguments to expresso functions are checked at runtime. On error an assertion fails and the program aborts. Together with a standard debugger this will hopefully help in finding your bugs easily.

Beware that the runtime checks activated by the EXPRESSO_DEBUG macro will decrease efficiency drastically. It is only meant for use while developing or debugging.

Also beware that compiling only some files with EXPRESSO_DEBUG may sometimes produce strange results because inlined functions may be defined differently in different object files. It is recommended not to link together files compiled with different EXPRESSO_DEBUG macro settings.

1.4.12 Compilation errors

In the design of expresso emphasis has been made to maximize the compiler's ability to discover typos and other errors at compile time wherever possible. On the other hand it is impossible to control how each compiler reports a compile error. In many cases a syntax error can produce a very long and cryptic compiler error. The reason is the heavy use of recursively defined template classes needed to attain the features and high performance that expresso provides. It is often quite impossible to understand what the problem is by just looking at the compiler output.

The recommended way of dealing with this is to just look at the compiler output to determine which line in your code caused the error, then look in your code at that line to see what the problem could be.

The good part is that (in contrast to fortran and other less type-safe languages) if your code compiles there is a very good chance that it actually does what you expected.

Author:

Oskar Enoksson, 2004

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 3

Class Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

Namespace Documentation

5.1 esso Namespace Reference

Namespace for fast arrays for numerical computation.

Classes

- class **Array< N, Fix< T0 > >**

  Multidimensional, efficient array class for large data sets.

- class **Array**

  Multidimensional, efficient array class for large data sets.

- class **Ix**

  Represents a small, fast vector.

- class **Permutation**

  Represents a permutation of dimensions.

Typedefs

- typedef int **IXTYPE**
- typedef short **DIMT**

Functions

Array assignment operations

- template<int N, class DType, class ST>
  const **Array**< N, DType > & **operator**<<= (const **Array**< N, DType > &dst, const ST &src)
- template<int N, class DType, class SType>
  const **Array**< N, DType > & **operator**<<= (const **Array**< N, DType > &dst, const **Array**< N, SType > &src)

- template<int N, class DType, class ST>
  const **Array**< N, DType > & **operator+=** (const **Array**< N, DType > &dst, const ST &src)
- template<int N, class DType, class SType>
  const **Array**< N, DType > & **operator+=** (const **Array**< N, DType > &dst, const **Array**< N, SType > &src)
- template<int N, class DType, class ST>
  const **Array**< N, DType > & **operator-=** (const **Array**< N, DType > &dst, const ST &src)
- template<int N, class DType, class SType>
  const **Array**< N, DType > & **operator-=** (const **Array**< N, DType > &dst, const **Array**< N, SType > &src)
- template<int N, class DType, class ST>
  const **Array**< N, DType > & **operator*=** (const **Array**< N, DType > &dst, const ST &src)
- template<int N, class DType, class SType>
  const **Array**< N, DType > & **operator*=** (const **Array**< N, DType > &dst, const **Array**< N, SType > &src)
- template<int N, class DType, class ST>
  const **Array**< N, DType > & **operator/=** (const **Array**< N, DType > &dst, const ST &src)
- template<int N, class DType, class SType>
  const **Array**< N, DType > & **operator/=** (const **Array**< N, DType > &dst, const **Array**< N, SType > &src)

Ix functions

- template<int N, class T>
  Ix< N, T > **operator-** (const **Ix**< N, T > &ix)
- template<int N, class T>
  Ix< N, T > **operator+** (const **Ix**< N, T > &ix, T i)
- template<int N, class T>
  Ix< N, T > **operator+** (const **Ix**< N, T > &ix, const **Ix**< N, T > &ixx)
- template<int N, class T>
  Ix< N, T > **operator+** (T i, const **Ix**< N, T > &ix)
- template<int N, class T>
  Ix< N, T > **operator-** (const **Ix**< N, T > &ix, T i)
- template<int N, class T>
  Ix< N, T > **operator-** (const **Ix**< N, T > &ix, const **Ix**< N, T > &ixx)
- template<int N, class T>
  Ix< N, T > **operator-** (T i, const **Ix**< N, T > &ix)
- template<int N, class T>
  Ix< N, T > **operator*** (const **Ix**< N, T > &ix, T i)
- template<int N, class T>
  Ix< N, T > **operator*** (T i, const **Ix**< N, T > &ix)
- template<int N, class T>
  Ix< N, T > **operator/** (const **Ix**< N, T > &ix, T i)
- template<int N, class T>
  Ix< N, T > **operator/** (T i, const **Ix**< N, T > &ix)

MPI support

- template<int N, class Type>
  MPI::Datatype **CreateMPIPPArray** (const **Array**< N, Type > &array)
- template<int N, class Type>
  MPI_Datatype **CreateMPIArray** (const **Array**< N, Type > &array)

Arithmetic Array operations

- template<int N, class TypeL, class TypeR>
  Array< N, Bop< cAdd, TypeL, TypeR > > **operator+** (const **Array**< N, TypeL > &l, const **Array**< N, TypeR > &r)
- template<int N, class TypeL, class TR>
  Array< N, Bop< cAdd, TypeL, Scalar< TR > > > **operator+** (const **Array**< N, TypeL > &l, const TR &r)
- template<int N, class TL, class TypeR>
  Array< N, Bop< cAdd, Scalar< TL >, TypeR > > **operator+** (const TL &l, const **Array**< N, TypeR > &r)
- template<int N, class TypeL, class TypeR>
  Array< N, Bop< cSub, TypeL, TypeR > > **operator-** (const **Array**< N, TypeL > &l, const **Array**< N, TypeR > &r)
- template<int N, class TypeL, class TR>
  Array< N, Bop< cSub, TypeL, Scalar< TR > > > **operator-** (const **Array**< N, TypeL > &l, const TR &r)
- template<int N, class TL, class TypeR>
  Array< N, Bop< cSub, Scalar< TL >, TypeR > > **operator-** (const TL &l, const **Array**< N, TypeR > &r)
- template<int N, class TypeL, class TypeR>
  Array< N, Bop< cMul, TypeL, TypeR > > **operator*** (const **Array**< N, TypeL > &l, const **Array**< N, TypeR > &r)
- template<int N, class TypeL, class TR>
  Array< N, Bop< cMul, TypeL, Scalar< TR > > > **operator*** (const **Array**< N, TypeL > &l, const TR &r)
- template<int N, class TL, class TypeR>
  Array< N, Bop< cMul, Scalar< TL >, TypeR > > **operator*** (const TL &l, const **Array**< N, TypeR > &r)
- template<int N, class TypeL, class TypeR>
  Array< N, Bop< cDiv, TypeL, TypeR > > **operator/** (const **Array**< N, TypeL > &l, const **Array**< N, TypeR > &r)
- template<int N, class TypeL, class TR>
  Array< N, Bop< cDiv, TypeL, Scalar< TR > > > **operator/** (const **Array**< N, TypeL > &l, const TR &r)
- template<int N, class TL, class TypeR>
  Array< N, Bop< cDiv, Scalar< TL >, TypeR > > **operator/** (const TL &l, const **Array**< N, TypeR > &r)
- template<int N, class TypeL, class TypeR>
  Array< N, Bop< cMax, TypeL, TypeR > > **Max** (const **Array**< N, TypeL > &l, const **Array**< N, TypeR > &r)
- template<int N, class TypeL, class TR>
  Array< N, Bop< cMax, TypeL, Scalar< TR > > > **Max** (const **Array**< N, TypeL > &l, const TR &r)
- template<int N, class TL, class TypeR>
  Array< N, Bop< cMax, Scalar< TL >, TypeR > > **Max** (const TL &l, const **Array**< N, TypeR > &r)
- template<int N, class TypeL, class TypeR>
  Array< N, Bop< cMin, TypeL, TypeR > > **Min** (const **Array**< N, TypeL > &l, const **Array**< N, TypeR > &r)
- template<int N, class TypeL, class TR>
  Array< N, Bop< cMin, TypeL, Scalar< TR > > > **Min** (const **Array**< N, TypeL > &l, const TR &r)
- template<int N, class TL, class TypeR>
  Array< N, Bop< cMin, Scalar< TL >, TypeR > > **Min** (const TL &l, const **Array**< N, TypeR > &r)

- template<int N, class TypeS>
  Array< N, Uop< cNeg, TypeS > > **operator-** (const **Array**< N, TypeS > &s)
- template<int N, class TypeS>
  Array< N, Uop< cSqrt, TypeS > > **Sqrt** (const **Array**< N, TypeS > &s)
- template<int N, class TypeS>
  Array< N, Uop< cSqr, TypeS > > **Sqr** (const **Array**< N, TypeS > &s)
- template<int N, class TypeS>
  Array< N, Uop< cAbs, TypeS > > **Abs** (const **Array**< N, TypeS > &s)
- template<int N, class TypeS>
  Array< N-1, Rop< cSum, TypeS > > **Sum** (const **Array**< N, TypeS > &s)
- template<class TypeS>
  RPromote< cSum, typename **Array**< 1, TypeS >::T >::T **Sum** (const **Array**< 1, TypeS > &src0)
- template<int N, class TypeS>
  Array< N, TypeS >::T **Sumall** (const **Array**< N, TypeS > &src)
- template<class TypeS>
  Array< 1, TypeS >::T **Sumall** (const **Array**< 1, TypeS > &src)
- template<int N, class TypeS>
  Array< N-1, Rop< cMax, TypeS > > **Max** (const **Array**< N, TypeS > &s)
- template<class TypeS>
  RPromote< cMax, typename **Array**< 1, TypeS >::T >::T **Max** (const **Array**< 1, TypeS > &src0)
- template<int N, class TypeS>
  Array< N, TypeS >::T **Maxall** (const **Array**< N, TypeS > &src)
- template<class TypeS>
  Array< 1, TypeS >::T **Maxall** (const **Array**< 1, TypeS > &src)
- template<int N, class TypeS>
  Array< N-1, Rop< cMin, TypeS > > **Min** (const **Array**< N, TypeS > &s)
- template<class TypeS>
  RPromote< cMin, typename **Array**< 1, TypeS >::T >::T **Min** (const **Array**< 1, TypeS > &src0)
- template<int N, class TypeS>
  Array< N, TypeS >::T **Minall** (const **Array**< N, TypeS > &src)
- template<class TypeS>
  Array< 1, TypeS >::T **Minall** (const **Array**< 1, TypeS > &src)

Logical Array operations

- template<int N, class Type>
  Array< N, Type > **Shift** (const **Array**< N, Type > &src, **DIMT** n, **IXTYPE** i)
- template<int N, class Type>
  Array< N, Type > **Shift** (const **Array**< N, Type > &src, const **Ix**< N > &ix)
- template<int N, class Type>
  Array< N, Type > **Reverse** (const **Array**< N, Type > &src, **DIMT** n)
- template<int N, class Type>
  Array< N, Type > **Reverse** (const **Array**< N, Type > &src, const **Ix**< N, bool > rx)
- template<int N, class Type>
  Array< N, Type > **Flip** (const **Array**< N, Type > &src, **DIMT** n)
- template<int N, class Type>
  Array< N, Type > **Flip** (const **Array**< N, Type > &src, const **Ix**< N, bool > &rx)
- template<int N, class Type>
  Array< N, Type > **Restride** (const **Array**< N, Type > &src, **DIMT** n, **IXTYPE** s)
- template<int N, class Type>
  Array< N, Type > **Restride** (const **Array**< N, Type > &src, const **Ix**< N > &sx)
- template<int N, class Type>
  Array< N, Type > **RestrictL** (const **Array**< N, Type > &src, **DIMT** n, **IXTYPE** l0)

- template<int N, class Type>
  Array< N, Type > **RestrictU** (const **Array**< N, Type > &src, **DIMT** n, **IXTYPE** u0)
- template<int N, class Type>
  Array< N, Type > **Restrict** (const **Array**< N, Type > &src, **DIMT** n, **IXTYPE** l0, **IXTYPE** u0)
- template<int N, class Type>
  Array< N, Type > **RestrictL** (const **Array**< N, Type > &src, const **Ix**< N > &l0)
- template<int N, class Type>
  Array< N, Type > **RestrictU** (const **Array**< N, Type > &src, const **Ix**< N > &u0)
- template<int N, class Type>
  Array< N, Type > **Restrict** (const **Array**< N, Type > &src, const **Ix**< N > &l0, const **Ix**< N > &u0)
- template<int N, class Type>
  Array< N, Type > **Permute** (const **Array**< N, Type > &src, **DIMT** n1, **DIMT** n2)
- template<int N, class Type>
  Array< N, Type > **Permute** (const **Array**< N, Type > &src, const **Permutation**< N > &p)
- template<int N, class Type>
  Array< N-1, Type > **Rmdim** (const **Array**< N, Type > &src, **DIMT** n, **IXTYPE** i)
- template<int N, class Type>
  Array< N+1, Type > **Extend** (const **Array**< N, Type > &src, **IXTYPE** l0, **IXTYPE** u0)
- template<int N, class Type>
  Array< N, Fix< typename **Array**< N, Type >::T > > **Clone** (const **Array**< N, Type > &src)
- template<int N, class TypeL, class TypeR>
  bool **Matches** (const **Array**< N, TypeL > &l, const **Array**< N, TypeR > &r)

Stream functions

- template<int N, class Type>
  std::ostream & **WriteUfmt** (std::ostream &stream, const **Array**< N, Type > &src)
- template<int N, class Type>
  std::ostream & **WriteFmt** (std::ostream &stream, const **Array**< N, Type > &src)
- template<int N, class Type>
  std::istream & **ReadUfmt** (std::istream &stream, const **Array**< N, Type > &dst)
- template<int N, class Type>
  std::istream & **ReadFmt** (std::istream &stream, const **Array**< N, Type > &dst)
- template<int N, class T>
  std::istream & **ReadFmt** (std::istream &stream, **Ix**< N, T > &dst)
- template<int N, class T>
  std::istream & **ReadUfmt** (std::istream &stream, **Ix**< N, T > &dst)
- template<int N, class T>
  std::ostream & **WriteFmt** (std::ostream &stream, const **Ix**< N, T > &src)
- template<int N, class T>
  std::ostream & **WriteUfmt** (std::ostream &stream, const **Ix**< N, T > &src)
- template<int N, class Type>
  std::ostream & **operator**<< (std::ostream &stream, const **Array**< N, Type > &src)
- template<int N, class Type>
  std::istream & **operator**>> (std::istream &stream, const **Array**< N, Type > &dst)
- template<int N, class Type>
  bool **Clone** (std::istream &stream, **Array**< N, Type > &dst)

5.1.1 Detailed Description

Namespace for fast arrays for numerical computation.

5.1.2 Typedef Documentation

5.1.2.1 typedef short esso::DIMT

Type used to indicate **Array** (p. 41) dimension

5.1.2.2 typedef int esso::IXTYPE

Type used to index **Array** (p. 41) objects

5.1.3 Function Documentation

5.1.3.1 template<int N, class TypeS> Array<N,Uop<cAbs,TypeS> > esso::Abs (const Array< N, TypeS > & *s*) `[inline]`

Absolute value

- *s* Source array object

  Returns:

  Array (p. 41) with each element equal to the absolute value of that from *s*

5.1.3.2 template<int N, class Type> bool esso::Clone (std::istream & *stream*, Array< N, Type > & *dst*) `[inline]`

Realloc **Array** (p. 41) from contents of formatted stream and read all elements from the stream.

- *stream* input stream
- *dst* Destination array

  Returns:

  true on success, or false if operation failed

  See also:

  Realloc(std::istream &, Array<N,Type> &)
  operator ><(std::istream &, const **Array<N,Type>** (p. 41) &)

References ReadFmt(), and esso::Array< N, T0 >::Realloc().

5.1.3.3 template<int N, class Type> Array<N,Fix<typename Array<N,Type>::T> > esso::Clone (const Array< N, Type > & *src*) `[inline]`

- *src* Source array

  Returns:

  new array that is a copy of all elements in *src*

  See also:

  Array::Clone (p. 46)

References esso::Array< N, T0 >::Clone().

Referenced by esso::Array< N, Fix< T0 > >::Clone().

5.1.3.4 template<int N, class Type> MPI_Datatype esso::CreateMPIArray (const Array< N, Type > & *array*) `[inline]`

Creates an MPI datatype corresponding to the given array The returned object should be freed when no longer needed

References esso::Array< N, T0 >::IsNull(), esso::Array< N, T0 >::Len(), and esso::Array< N, T0 >::Stride().

5.1.3.5 template<int N, class Type> MPI::Datatype esso::CreateMPIPPArray (const Array< N, Type > & *array*) `[inline]`

Creates an MPI++ datatype corresponding to the given array The returned object should be freed when no longer needed

References esso::Array< N, T0 >::IsNull(), esso::Array< N, T0 >::Len(), and esso::Array< N, T0 >::Stride().

5.1.3.6 template<int N, class Type> Array<N+1,Type> esso::Extend (const Array< N, Type > & *src*, IXTYPE *l0*, IXTYPE *u0*) `[inline]`

- *src* Source array
- *l0* Lower index bound for new dimension
- *u0* Upper index bound for new dimension

Returns:

new array refering to the same data as *src* with a new dummy dimension inserted at dimension 0

See also:

Array::Permute (p. 47)

5.1.3.7 template<int N, class Type> Array<N,Type> esso::Flip (const Array< N, Type > & *src*, const Ix< N, bool > & *rx*) `[inline]`

- *src* Source array
- *rx* boolean vector indicating which dimensions to flip

Returns:

new flipped array refering to the same data as *src*

See also:

Array::Flip (p. 48)

5.1.3.8 template<int N, class Type> Array<N,Type> esso::Flip (const Array< N, Type > & *src*, DIMT *n*) `[inline]`

- *src* Source array
- *n* dimension to flip

Returns:

new flipped array refering to the same data as *src*

See also:

Array::Flip (p. 48)

5.1.3.9 template<int N, class TypeL, class TypeR> bool esso::Matches (const Array< N, TypeL > & *l*, const Array< N, TypeR > & *r*) `[inline]`

- *l* first array to compare
- *r* second array to compare

Returns:

true iff the index bounds are identical for *l* and *r*

References esso::Array< N, T0 >::L(), and esso::Array< N, T0 >::U().

5.1.3.10 template<class TypeS> RPromote<cMax,typename Array<1,TypeS>::T>::T esso::Max (const Array< 1, TypeS > & *src0*) `[inline]`

- *src* Source array

Returns:

Maximum of all elements in *src*

5.1.3.11 template<int N, class TypeS> Array<N-1,Rop<cMax,TypeS> > esso::Max (const Array< N, TypeS > & *s*) `[inline]`

- *src* Source array

Returns:

Array (p. 41) where each element is the maximum of all elements in *src* along dimension 0

5.1.3.12 template<int N, class TL, class TypeR> Array<N,Bop<cMax,Scalar<TL>,TypeR > > esso::Max (const TL & *l*, const Array< N, TypeR > & *r*) `[inline]`

- *l* Left scalar operand
- *r* Right **Array** (p. 41) operand

Returns:

Array (p. 41) where each element refers to the maximum of the corresponding elements in l and r

5.1.3.13 template<int N, class TypeL, class TR> Array<N,Bop<cMax,TypeL,Scalar<TR> > > esso::Max (const Array< N, TypeL > & *l*, const TR & *r*) `[inline]`

- *l* Left **Array** (p. 41) operand
- *r* Right scalar operand

Returns:

Array (p. 41) where each element refers to the maximum of the corresponding elements in l and r

5.1.3.14 template<int N, class TypeL, class TypeR> Array<N,Bop<cMax,TypeL,TypeR> > esso::Max (const Array< N, TypeL > & *l*, const Array< N, TypeR > & *r*) `[inline]`

- *l* Left **Array** (p. 41) operand
- *r* Right **Array** (p. 41) operand

Returns:

Array (p. 41) where each element refers to the maximum of the corresponding elements in l and r

Referenced by Maxall().

5.1.3.15 template<int N, class TypeS> Array<N,TypeS>::T esso::Maxall (const Array< N, TypeS > & *src*) `[inline]`

- *src* Source array

Returns:

Maximum value of all elements in array

References Max().

5.1.3.16 template<class TypeS> RPromote<cMin,typename Array<1,TypeS>::T>::T esso::Min (const Array< 1, TypeS > & *src0*) `[inline]`

- *src* Source array

Returns:

Minimum of all elements in *src*

5.1.3.17 template<int N, class TypeS> Array<N-1,Rop<cMin,TypeS> > esso::Min (const Array< N, TypeS > & *s*) [inline]

- *src* Source array

Returns:

Array (p. 41) where each element is the minimum of all elements in *src* along dimension 0

5.1.3.18 template<int N, class TL, class TypeR> Array<N,Bop<cMin,Scalar<TL>,TypeR > > esso::Min (const TL & *l*, const Array< N, TypeR > & *r*) [inline]

- *l* Left scalar operand
- *r* Right **Array** (p. 41) operand

Returns:

Array (p. 41) where each element refers to the minimum of the corresponding elements in *l* and *r*

5.1.3.19 template<int N, class TypeL, class TR> Array<N,Bop<cMin,TypeL,Scalar<TR> > > esso::Min (const Array< N, TypeL > & *l*, const TR & *r*) [inline]

- *l* Left **Array** (p. 41) operand
- *r* Right scalar operand

Returns:

Array (p. 41) where each element refers to the minimum of the corresponding elements in *l* and *r*

5.1.3.20 template<int N, class TypeL, class TypeR> Array<N,Bop<cMin,TypeL,TypeR> > esso::Min (const Array< N, TypeL > & *l*, const Array< N, TypeR > & *r*) [inline]

- *l* Left **Array** (p. 41) operand
- *r* Right **Array** (p. 41) operand

Returns:

Array (p. 41) where each element refers to the minimum of the corresponding elements in *l* and *r*

Referenced by Minall().

5.1.3.21 template<int N, class TypeS> Array<N,TypeS>::T esso::Minall (const Array< N, TypeS > & *src*) [inline]

- *src* Source array

Returns:

Minimum value of all elements in array

References Min().

5.1.3.22 template<int N, class TL, class TypeR> Array<N,Bop<cMul,Scalar<TL>,TypeR > > esso::operator* (const TL & *l*, const Array< N, TypeR > & *r*) [inline]

- *l* Left scalar operand
- *r* Right **Array** (p. 41) operand

Returns:

Array (p. 41) where each element refers to the product of the corresponding elements in *l* and *r*

5.1.3.23 template<int N, class TypeL, class TR> Array<N,Bop<cMul,TypeL,Scalar<TR> > > esso::operator* (const Array< N, TypeL > & *l*, const TR & *r*) [inline]

- *l* Left **Array** (p. 41) operand
- *r* Right scalar operand

Returns:

Array (p. 41) where each element refers to the product of the corresponding elements in *l* and *r*

5.1.3.24 template<int N, class TypeL, class TypeR> Array<N,Bop<cMul,TypeL,TypeR> > esso::operator* (const Array< N, TypeL > & *l*, const Array< N, TypeR > & *r*) [inline]

- *l* Left **Array** (p. 41) operand
- *r* Right **Array** (p. 41) operand

Returns:

Array (p. 41) where each element refers to the product of the corresponding elements in *l* and *r*

5.1.3.25 template<int N, class T> Ix<N,T> esso::operator* (T *i*, const Ix< N, T > & *ix*) [inline]

Product scalar * object

5.1.3.26 template<int N, class T> Ix<N,T> esso::operator* (const Ix< N, T > & *ix*, T *i*) `[inline]`

Product object * scalar

5.1.3.27 template<int N, class DType, class SType> const Array<N,DType>& esso::operator*= (const Array< N, DType > & *dst*, const Array< N, SType > & *src*) `[inline]`

Scalar multiplication assignment. Multiply scalar *src* to elements of *dst*

5.1.3.28 template<int N, class DType, class ST> const Array<N,DType>& esso::operator*= (const Array< N, DType > & *dst*, const ST & *src*) `[inline]`

Multiplication assignment. Multipllies elements of *src* to elements of *dst* Operates on the intersection of index range for *src* and *dst*

5.1.3.29 template<int N, class TL, class TypeR> Array<N,Bop<cAdd,Scalar<TL>,TypeR > > esso::operator+ (const TL & *l*, const Array< N, TypeR > & *r*) `[inline]`

- *l* Left scalar operand
- *r* Right **Array** (p. 41) operand

Returns:

Array (p. 41) where each element refers to the sum of the corresponding elements in *l* and *r*

5.1.3.30 template<int N, class TypeL, class TR> Array<N,Bop<cAdd,TypeL,Scalar<TR> > > esso::operator+ (const Array< N, TypeL > & *l*, const TR & *r*) `[inline]`

- *l* Left **Array** (p. 41) operand
- *r* Right scalar operand

Returns:

Array (p. 41) where each element refers to the sum of the corresponding elements in *l* and *r*

5.1.3.31 template<int N, class TypeL, class TypeR> Array<N,Bop<cAdd,TypeL,TypeR> > esso::operator+ (const Array< N, TypeL > & *l*, const Array< N, TypeR > & *r*) `[inline]`

- *l* Left **Array** (p. 41) operand
- *r* Right **Array** (p. 41) operand

Returns:

Array (p. 41) where each element refers to the sum of the corresponding elements in *l* and *r*

5.1.3.32 template<int N, class T> Ix<N,T> esso::operator+ (T *i*, const Ix< N, T > & *ix*) `[inline]`

Addition scalar + object

5.1.3.33 template<int N, class T> Ix<N,T> esso::operator+ (const Ix< N, T > & *ix*, const Ix< N, T > & *ixx*) `[inline]`

Addition

5.1.3.34 template<int N, class T> Ix<N,T> esso::operator+ (const Ix< N, T > & *ix*, T *i*) `[inline]`

Addition, object + scalar

5.1.3.35 template<int N, class DType, class SType> const Array<N,DType>& esso::operator+= (const Array< N, DType > & *dst*, const Array< N, SType > & *src*) `[inline]`

Scalar addition assignment. Add scalar *src* to elements of *dst*

5.1.3.36 template<int N, class DType, class ST> const Array<N,DType>& esso::operator+= (const Array< N, DType > & *dst*, const ST & *src*) `[inline]`

Addition assignment. Add elements of *src* to elements of *dst* Operates on the intersection of index range for *src* and *dst*

5.1.3.37 template<int N, class TypeS> Array<N,Uop<cNeg,TypeS> > esso::operator- (const Array< N, TypeS > & *s*) `[inline]`

Unary negation

- *s* Source array object

Returns:

Array (p. 41) with each element from *s* negated

5.1.3.38 template<int N, class TL, class TypeR> Array<N,Bop<cSub,Scalar<TL>,TypeR > > esso::operator- (const TL & *l*, const Array< N, TypeR > & *r*) `[inline]`

- *l* Left scalar operand

- *r* Right **Array** (p. 41) operand

Returns:

Array (p. 41) where each element refers to the difference of the corresponding elements in *l* and *r*

5.1.3.39 template<int N, class TypeL, class TR> Array<N,Bop<cSub,TypeL,Scalar<TR> > > esso::operator- (const Array< N, TypeL > & *l*, const TR & *r*) `[inline]`

- *l* Left **Array** (p. 41) operand
- *r* Right scalar operand

Returns:

Array (p. 41) where each element refers to the difference of the corresponding elements in *l* and *r*

5.1.3.40 template<int N, class TypeL, class TypeR> Array<N,Bop<cSub,TypeL,TypeR> > esso::operator- (const Array< N, TypeL > & *l*, const Array< N, TypeR > & *r*) `[inline]`

- *l* Left **Array** (p. 41) operand
- *r* Right **Array** (p. 41) operand

Returns:

Array (p. 41) where each element refers to the difference of the corresponding elements in *l* and *r*

5.1.3.41 template<int N, class T> Ix<N,T> esso::operator- (T *i*, const Ix< N, T > & *ix*) `[inline]`

Subtraction scalar - object

5.1.3.42 template<int N, class T> Ix<N,T> esso::operator- (const Ix< N, T > & *ix*, const Ix< N, T > & *ixx*) `[inline]`

Subtraction

5.1.3.43 template<int N, class T> Ix<N,T> esso::operator- (const Ix< N, T > & *ix*, T *i*) `[inline]`

Subtraction object - scalar

5.1.3.44 template<int N, class T> Ix<N,T> esso::operator- (const Ix< N, T > & *ix*) `[inline]`

Unary negation

5.1.3.45 template<int N, class DType, class SType> const Array<N,DType>& esso::operator-= (const Array< N, DType > & *dst*, const Array< N, SType > & *src*) [inline]

Scalar subtraction assignment. Subtract scalar *src* from elements of *dst*

5.1.3.46 template<int N, class DType, class ST> const Array<N,DType>& esso::operator-= (const Array< N, DType > & *dst*, const ST & *src*) [inline]

Subtraction assignment. Subtract elements of *src* from elements of *dst* Operates on the intersection of index range for *src* and *dst*

5.1.3.47 template<int N, class TL, class TypeR> Array<N,Bop<cDiv,Scalar<TL>,TypeR > > esso::operator/ (const TL & *l*, const Array< N, TypeR > & *r*) [inline]

- *l* Left scalar operand
- *r* Right **Array** (p. 41) operand

Returns:

Array (p. 41) where each element refers to the quotient of the corresponding elements in *l* and *r*

5.1.3.48 template<int N, class TypeL, class TR> Array<N,Bop<cDiv,TypeL,Scalar<TR> > > esso::operator/ (const Array< N, TypeL > & *l*, const TR & *r*) [inline]

- *l* Left **Array** (p. 41) operand
- *r* Right scalar operand

Returns:

Array (p. 41) where each element refers to the quotient of the corresponding elements in *l* and *r*

5.1.3.49 template<int N, class TypeL, class TypeR> Array<N,Bop<cDiv,TypeL,TypeR> > esso::operator/ (const Array< N, TypeL > & *l*, const Array< N, TypeR > & *r*) [inline]

- *l* Left **Array** (p. 41) operand
- *r* Right **Array** (p. 41) operand

Returns:

Array (p. 41) where each element refers to the quotient of the corresponding elements in *l* and *r*

5.1.3.50 template<int N, class T> Ix<N,T> esso::operator/ (T *i*, const Ix< N, T > & *ix*) [inline]

Quotient scalar / object

5.1.3.51 template<int N, class T> Ix<N,T> esso::operator/ (const Ix< N, T > & *ix*, T *i*) [inline]

Quotient object / scalar

5.1.3.52 template<int N, class DType, class SType> const Array<N,DType>& esso::operator/= (const Array< N, DType > & *dst*, const Array< N, SType > & *src*) [inline]

Quotient assignment. Divides elements of *dst* with elements of *src* Operates on the intersection of index range for *src* and *dst*

5.1.3.53 template<int N, class DType, class ST> const Array<N,DType>& esso::operator/= (const Array< N, DType > & *dst*, const ST & *src*) [inline]

Scalar quotient assignment. Divide elements of *dst* with scalar *src*

5.1.3.54 template<int N, class Type> std::ostream& esso::operator<< (std::ostream & *stream*, const Array< N, Type > & *src*) [inline]

Ascii stream store operation

Output format is

- N
- LEN(0) L(0)
- LEN(1) L(1)
- ...
- LEN(N-1) L(N-1)
- DATA (LINE 1)
- DATA (LINE 2)
- ...

where `N` is the array rank, `LEN(K)` is the length along dimension `K` and `L(K)` is the first index in dimension `K`. Each line in `DATA` is along dimension 0, and the other dimensions are traversed in increasing order, so that the outermost loop is over dimension `N-1`.

- *stream* output stream
- *src* Source array

Returns:

Reference to output stream *stream*

References WriteFmt().

5.1.3.55 template<int N, class DType, class SType> const Array<N,DType>& esso::operator<<= (const Array< N, DType > & *dst*, const Array< N, SType > & *src*) `[inline]`

Elementwise copy

Use this operator for template expression assignment i.e. you may copy a single array into another as

`array1 <<= array2;`

or you may compute expressions containing arrays and/or scalars and put the result into an array as

`array1 <<= array2+array3*scalar1-array1;`

The assignment is done over **the intersection** of the index bounds of all array objects involved. The index bounds of *dst* are unchanged by this operation.

- *dst* Destination array object
- *src* Source array object

Returns:

reference to destination array

5.1.3.56 template<int N, class DType, class ST> const Array<N,DType>& esso::operator<<= (const Array< N, DType > & *dst*, const ST & *src*) `[inline]`

Elementwise copy Use this operator to fill an array with a scalar value

- *dst* Destination array object
- *src* Source scalar

Returns:

reference to destination array

5.1.3.57 template<int N, class Type> std::istream& esso::operator>> (std::istream & *stream*, const Array< N, Type > & *dst*) `[inline]`

Ascii stream read operation

The elements of the array are read from a file that is assumed to have the format as written by operator <<

If the array index bounds do not match the array found in the file the array elements are unchanged and the fail flag is set for the stream object.

- *stream* input stream
- *src* Destination array

Returns:

true on success, or false if operation failed
Reference to input stream *stream*

References esso::Array< N, T0 >::L(), ReadFmt(), and esso::Array< N, T0 >::U().

5.1.3.58 template<int N, class Type> Array<N,Type> esso::Permute (const Array< N, Type > & *src*, const Permutation< N > & *p*) `[inline]`

- *src* Source array
- *p* permutation to apply to array dimensions

Returns:

new permuted array refering to the same data as *src*

See also:

Array::Permute (p. 47)

5.1.3.59 template<int N, class Type> Array<N,Type> esso::Permute (const Array< N, Type > & *src*, DIMT *n1*, DIMT *n2*) `[inline]`

- *src* Source array
- *n1 n2* Dimensions to swap

Returns:

new permuted array refering to the same data as *src*

See also:

Array::Permute (p. 47)

5.1.3.60 template<int N, class T> std::istream& esso::ReadFmt (std::istream & *stream*, Ix< N, T > & *dst*) `[inline]`

Formated read of **Ix** (p. 58) object

5.1.3.61 template<int N, class Type> std::istream& esso::ReadFmt (std::istream & *stream*, const Array< N, Type > & *dst*) `[inline]`

Formatted stream read

Simply reads all elements from the *stream*. The dimensions are traversed in increasing order, so that the outermost loop is over dimension `N-1`.

- *stream* output stream

- *src* Source array

  Returns:

  Reference to input stream *stream*

Referenced by Clone(), and operator>>().

5.1.3.62 template<int N, class T> std::istream& esso::ReadUfmt (std::istream & *stream*, Ix< N, T > & *dst*) [inline]

Unformated read of **Ix** (p. 58) object

5.1.3.63 template<int N, class Type> std::istream& esso::ReadUfmt (std::istream & *stream*, const Array< N, Type > & *dst*) [inline]

Unformatted stream read

Simply reads all elements from the *stream*. The dimensions are traversed in increasing order, so that the outermost loop is over dimension `N-1`.

- *stream* output stream
- *src* Source array

  Returns:

  Reference to input stream *stream*

5.1.3.64 template<int N, class Type> Array<N,Type> esso::Restrict (const Array< N, Type > & *src*, const Ix< N > & *l0*, const Ix< N > & *u0*) [inline]

- *src* Source array
- *l0* vector of new lower index bounds for each dimension
- *u0* vector of new upper index bounds for each dimension

  Returns:

  new restricted array refering to the same data as *src*

  See also:

  Array::Restrict (p. 47)

References Restrict().

5.1.3.65 template<int N, class Type> Array<N,Type> esso::Restrict (const Array< N, Type > & *src*, DIMT *n*, IXTYPE *l0*, IXTYPE *u0*) [inline]

- *src* Source array
- *n* dimension to restrict

- *l0* new lower index bound
- *u0* new upper index bound

Returns:

new restricted array refering to the same data as *src*

See also:

Array::Restrict (p. 47)

Referenced by Restrict().

5.1.3.66 template<int N, class Type> Array<N,Type> esso::RestrictL (const Array< N, Type > & *src*, const Ix< N > & *l0*) `[inline]`

- *src* Source array
- *l0* vector of new lower index bounds for each dimension

Returns:

new restricted array refering to the same data as *src*

See also:

Array::RestrictL (p. 48)

References RestrictL().

5.1.3.67 template<int N, class Type> Array<N,Type> esso::RestrictL (const Array< N, Type > & *src*, DIMT *n*, IXTYPE *l0*) `[inline]`

- *src* Source array
- *n* dimension to restrict
- *l0* new lower index bound

Returns:

new restricted array refering to the same data as *src*

See also:

Array::RestrictL (p. 48)

Referenced by RestrictL().

5.1.3.68 template<int N, class Type> Array<N,Type> esso::RestrictU (const Array< N, Type > & *src*, const Ix< N > & *u0*) `[inline]`

- *src* Source array
- *u0* vector of new upper index bounds for each dimension

Returns:

new restricted array refering to the same data as *src*

See also:

Array::RestrictU (p. 48)

References RestrictU().

5.1.3.69 template<int N, class Type> Array<N,Type> esso::RestrictU (const Array< N, Type > & *src*, DIMT *n*, IXTYPE *u0*) `[inline]`

- *src* Source array
- *n* dimension to restrict
- *u0* new upper index bound

Returns:

new restricted array refering to the same data as *src*

See also:

Array::RestrictU (p. 48)

Referenced by RestrictU().

5.1.3.70 template<int N, class Type> Array<N,Type> esso::Restride (const Array< N, Type > & *src*, const Ix< N > & *sx*) `[inline]`

- *src* Source array
- *sx* vector of restride multipliers for each dimension

Returns:

new restrided array refering to the same data as *src*

See also:

Array::Restride (p. 48)

5.1.3.71 template<int N, class Type> Array<N,Type> esso::Restride (const Array< N, Type > & *src*, DIMT *n*, IXTYPE *s*) `[inline]`

- *src* Source array
- *n* dimension to restride
- *s* restride multiplier

Returns:

new restrided array refering to the same data as *src*

See also:

Array::Restride (p. 48)

5.1.3.72 template<int N, class Type> Array<N,Type> esso::Reverse (const Array< N, Type > & *src*, const Ix< N, bool > *rx*) [inline]

- *src* Source array
- *rx* boolean vector indicating which dimensions to reverse

Returns:

new reversed array refering to the same data as *src*

See also:

Array::Reverse (p. 48)

5.1.3.73 template<int N, class Type> Array<N,Type> esso::Reverse (const Array< N, Type > & *src*, DIMT *n*) [inline]

- *src* Source array
- *n* dimension to reverse

Returns:

new reversed array refering to the same data as *src*

See also:

Array::Reverse (p. 48)

5.1.3.74 template<int N, class Type> Array<N-1,Type> esso::Rmdim (const Array< N, Type > & *src*, DIMT *n*, IXTYPE *i*) [inline]

- *src* Source array
- *n* Dimension to remove (how to cut a slice)
- *i* Index position along *n* (where to cut the slice)

Returns:

new array refering to a slice of the same data as *src*

See also:

Array::Rmdim (p. 46)

5.1.3.75 template<int N, class Type> Array<N,Type> esso::Shift (const Array< N, Type > & *src*, const Ix< N > & *ix*) [inline]

- *src* Source array
- *ix* vector of index positions to shift for each dimension

Returns:

new shifted array refering to the same data as *src*

See also:

Array::Shift (p. 47)

References Shift().

5.1.3.76 template<int N, class Type> Array<N,Type> esso::Shift (const Array< N, Type > & *src*, DIMT *n*, IXTYPE *i*) `[inline]`

- *src* Source array
- *n* dimension to shift
- *i* number of index positions to shift

Returns:

new shifted array refering to the same data as *src*

See also:

Array::Shift (p. 47)

Referenced by Shift(), and esso::Array< N, Fix< T0 > >::Shift().

5.1.3.77 template<int N, class TypeS> Array<N,Uop<cSqr,TypeS> > esso::Sqr (const Array< N, TypeS > & *s*) `[inline]`

Square (power of two)

- *s* Source array object

Returns:

Array (p. 41) with each element equal to the square of that from *s*

5.1.3.78 template<int N, class TypeS> Array<N,Uop<cSqrt,TypeS> > esso::Sqrt (const Array< N, TypeS > & *s*) `[inline]`

Square root

- *s* Source array object

Returns:

Array (p. 41) with each element square root of that from *s*

5.1.3.79 template<class TypeS> RPromote<cSum,typename Array<1,TypeS>::T>::T esso::Sum (const Array< 1, TypeS > & *src0*) `[inline]`

- *src* Source array

Returns:

Sum of all elements in *src*

5.1.3.80 template<int N, class TypeS> Array<N-1,Rop<cSum,TypeS> > esso::Sum (const Array< N, TypeS > & *s*) [inline]

- *src* Source array

Returns:

Array (p. 41) where each element is the sum of all elements in *src* along dimension 0

Referenced by Sumall().

5.1.3.81 template<int N, class TypeS> Array<N,TypeS>::T esso::Sumall (const Array< N, TypeS > & *src*) [inline]

- *src* Source array

Returns:

Sum of all elements in array

References Sum().

5.1.3.82 template<int N, class T> std::ostream& esso::WriteFmt (std::ostream & *stream*, const Ix< N, T > & *src*) [inline]

Formatted write of **Ix** (p. 58) object

5.1.3.83 template<int N, class Type> std::ostream& esso::WriteFmt (std::ostream & *stream*, const Array< N, Type > & *src*) [inline]

Formatted stream store

Simply dumps all elements into the *stream*. The dimensions are traversed in increasing order, so that the outermost loop is over dimension `N-1`.

- *stream* output stream
- *src* Source array

Returns:

Reference to output stream *stream*

Referenced by operator<<().

5.1.3.84 template<int N, class T> std::ostream& esso::WriteUfmt (std::ostream & *stream*, const Ix< N, T > & *src*) [inline]

Unformated write of **Ix** (p. 58) object

5.1.3.85 template<int N, class Type> std::ostream& esso::WriteUfmt (std::ostream & *stream*, const Array< N, Type > & *src*) [inline]

Unformatted stream store

Simply dumps all elements into the *stream*. The dimensions are traversed in increasing order, so that the outermost loop is over dimension `N-1`.

- *stream* output stream
- *src* Source array

Returns:

Reference to output stream *stream*

Chapter 6

Class Documentation

6.1 esso::Array< N, T0 > Class Template Reference

Multidimensional, efficient array class for large data sets.

```
#include <flex.h>
```

Public Member Functions

- T & **operator()** (**IXTYPE** i0) const
- T & **operator()** (**IXTYPE** i0, **IXTYPE** i1) const
- T & **operator()** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2) const
- T & **operator()** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2, **IXTYPE** i3) const
- T & **operator()** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2, **IXTYPE** i3, **IXTYPE** i4) const
- T & **operator()** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2, **IXTYPE** i3, **IXTYPE** i4, **IXTYPE** i5) const
- T & **operator[]** (const **Ix**< N > &ix) const

constructor

- **Array** ()
- **Array** (const **Ix**< N > &len0)
- **Array** (const **Ix**< N > &len0, const **Ix**< N > &l0)
- **Array** (const **Array** &src)
- template<class TypeS>
  Array (const **Array**< N, TypeS > &src)
- **Array** (const **Array**< N+1, Type > &src, **DIMT** n, **IXTYPE** i)
- **Array** (**IXTYPE** i0)
- **Array** (**IXTYPE** i0, **IXTYPE** i1)
- **Array** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2)
- **Array** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2, **IXTYPE** i3)
- **Array** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2, **IXTYPE** i3, **IXTYPE** i4)
- **Array** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2, **IXTYPE** i3, **IXTYPE** i4, **IXTYPE** i5)

const

- **IXTYPE L** (**DIMT** n) const
- **IXTYPE U** (**DIMT** n) const
- **IXTYPE Len** (**DIMT** n) const
- **IXTYPE Stride** (**DIMT** n) const
- T * **Origo** () const
- T * **Start** () const
- const MultiPtr< T > & **Data** () const
- bool **IsNull** () const

nonconst

- bool **Dealloc** ()
- bool **Realloc** (const **Ix**< N > &len0)
- bool **Realloc** (const **Ix**< N > &len0, const **Ix**< N > &l0)
- bool **Realloc** (const **Ix**< N > &len0, const **Ix**< N > &l0, const MultiPtr< T > &data0, T *origo0)
- template<class TypeS>
  bool **Realloc** (const **Array**< N, TypeS > &src)
- template<class TypeS>
  bool **Refer** (const **Array**< N, TypeS > &src)
- bool **Clone** ()
- template<class TypeS>
  bool **Clone** (const **Array**< N, TypeS > &src)
- template<class TypeS>
  Array & **Extend** (const **Array**< N-1, TypeS > &src, **IXTYPE** l0, **IXTYPE** u0)
- template<class TypeS>
  Array & **Rmdim** (const **Array**< N+1, TypeS > &src, **DIMT** n, **IXTYPE** i)
- template<class TypeS>
  Array & **operator=** (const **Array**< N, TypeS > &src)
- **Array** & **operator=** (const **Array** &src)
- **Array** & **Shift** (**DIMT** n, **IXTYPE** i)
- **Array** & **Shift** (const **Ix**< N > &ix)
- **Array** & **Permute** (**DIMT** n1, **DIMT** n2)
- **Array** & **Permute** (const **Permutation**< N > &nx)
- **Array** & **Restrict** (**DIMT** n, **IXTYPE** l0, **IXTYPE** u0)
- **Array** & **Restrict** (const **Ix**< N > &l0, const **Ix**< N > &u0)
- **Array** & **RestrictL** (**DIMT** n, **IXTYPE** l0)
- **Array** & **RestrictL** (const **Ix**< N > &l0)
- **Array** & **RestrictU** (**DIMT** n, **IXTYPE** u0)
- **Array** & **RestrictU** (const **Ix**< N > &u0)
- **Array** & **Flip** (**DIMT** n)
- **Array** & **Flip** (const **Ix**< N, bool > &r)
- **Array** & **Reverse** (**DIMT** n)
- **Array** & **Reverse** (const **Ix**< N, bool > &r)
- **Array** & **Restride** (**DIMT** n, **IXTYPE** s)
- **Array** & **Restride** (const **Ix**< N > &s)

6.1.1 Detailed Description

template<int N, class T0> class esso::Array< N, T0 >

Multidimensional, efficient array class for large data sets.

Use this array for efficient numerical computations or other applications where powerful and efficient arrays are needed for large data sets.

If efficiency is crusial, see also the specialization **Array** (p. 41)<N,Fix<T> > . It is faster but less flexible (cannot permute dimensions).

6.1.2 Constructor & Destructor Documentation

6.1.2.1 template<int N, class T0> esso::Array< N, T0 >::Array () [inline]

Default constructor. Initialize array with no data and index bounds zero

6.1.2.2 template<int N, class T0> esso::Array< N, T0 >::Array (const Ix< N > & *len0*) [inline]

Constructor with lower index bound set to zero

- len0 number of elements for each dimension

6.1.2.3 template<int N, class T0> esso::Array< N, T0 >::Array (const Ix< N > & *len0*, const Ix< N > & *l0*) [inline]

- len0 number of elements for each dimension
- l0 lower index bound

6.1.2.4 template<int N, class T0> esso::Array< N, T0 >::Array (const Array< N, T0 > & *src*) [inline]

Copy constructor Creates a reference to the same memory area as *src*

6.1.2.5 template<int N, class T0> template<class TypeS> esso::Array< N, T0 >::Array (const Array< N, TypeS > & *src*) [inline]

Copy constructor from another array type object If possible, create a reference to the same memory area as `src`. If this is not possible, create a new memory area and copy `src` elements

6.1.2.6 template<int N, class T0> esso::Array< N, T0 >::Array (const Array< N+1, Type > & *src*, DIMT *n*, IXTYPE *i*) [inline]

Constructor that creates reference to a slice along dimension `n` from higher-dimension array

References esso::Array< N, T0 >::L(), esso::Array< N, T0 >::Stride(), and esso::Array< N, T0 >::U().

6.1.2.7 template<int N, class T0> esso::Array< N, T0 >::Array (IXTYPE *i0*) [inline]

1-dimensional array of given length

6.1.2.8 template<int N, class T0> esso::Array< N, T0 >::Array (IXTYPE *i0*, IXTYPE *i1*) [inline]

2-dimensional array of given length

6.1.2.9 template<int N, class T0> esso::Array< N, T0 >::Array (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*) [inline]

3-dimensional array of given length

6.1.2.10 template<int N, class T0> esso::Array< N, T0 >::Array (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*, IXTYPE *i3*) [inline]

4-dimensional array of given length

6.1.2.11 template<int N, class T0> esso::Array< N, T0 >::Array (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*, IXTYPE *i3*, IXTYPE *i4*) [inline]

5-dimensional array of given length

6.1.2.12 template<int N, class T0> esso::Array< N, T0 >::Array (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*, IXTYPE *i3*, IXTYPE *i4*, IXTYPE *i5*) [inline]

6-dimensional array of given length

6.1.3 Member Function Documentation

6.1.3.1 template<int N, class T0> IXTYPE esso::Array< N, T0 >::L (DIMT *n*) const [inline]

Lower index bounds L(n) is the lowest index allowed on dimension `n`

Referenced by esso::Array< N, T0 >::Array(), esso::Array< N, T0 >::Extend(), esso::Matches(), esso::operator>>(), esso::Array< N, T0 >::Realloc(), esso::Array< N, T0 >::Refer(), and esso::Array< N, T0 >::Rmdim().

6.1.3.2 template<int N, class T0> IXTYPE esso::Array< N, T0 >::U (DIMT *n*) const [inline]

Upper index bounds U(n) is one more than the largest index allowed on dimension `n`

Referenced by esso::Array< N, T0 >::Array(), esso::Array< N, T0 >::Extend(), esso::Matches(), esso::operator>>(), esso::Array< N, T0 >::Realloc(), esso::Array< N, T0 >::Refer(), and esso::Array< N, T0 >::Rmdim().

6.1.3.3 template<int N, class T0> IXTYPE esso::Array< N, T0 >::Len (DIMT *n*) const [inline]

Convenience function giving the number of array elements along given dimension. Simply put, Len(n) is U(n) - L(n)

Referenced by esso::CreateMPIArray(), and esso::CreateMPIPPArray().

6.1.3.4 template<int N, class T0> T* esso::Array< N, T0 >::Start () const `[inline]`

Pointer to first valid element i.e. element at lowest index bound

Useful for passing pointer to fortran routine, but note that if the dimensions are permuted or reversed the data organization is not compatible with fortran.

6.1.3.5 template<int N, class T0> bool esso::Array< N, T0 >::IsNull () const `[inline]`

Returns:

true if this array has no data allocated

Referenced by esso::Array< N, T0 >::Clone(), esso::Array< N, Fix< T0 > >::Clone(), esso::CreateMPIArray(), and esso::CreateMPIPPArray().

6.1.3.6 template<int N, class T0> bool esso::Array< N, T0 >::Dealloc () `[inline]`

Release this object from it's element data and set index bounds to zero

6.1.3.7 template<int N, class T0> bool esso::Array< N, T0 >::Realloc (const Ix< N > & *len0*) `[inline]`

Release this object from it's element data and allocate a new data set with given size (lower index bounds zero)

Returns:

always returns true

Referenced by esso::Clone(), esso::Array< N, T0 >::Clone(), and esso::Array< N, Fix< T0 > >::Clone().

6.1.3.8 template<int N, class T0> bool esso::Array< N, T0 >::Realloc (const Ix< N > & *len0*, const Ix< N > & *l0*) `[inline]`

Release this object from it's element data and allocate a new data set with given size and given lower index bounds

Returns:

always returns true

6.1.3.9 template<int N, class T0> template<class TypeS> bool esso::Array< N, T0 >::Realloc (const Array< N, TypeS > & *src*) `[inline]`

Release this object from it's element data and allocate a new, uninitialized data set with given size and index bounds identical with those of `src`.

Returns:

true on success or false if operation failed

References esso::Array< N, T0 >::L(), and esso::Array< N, T0 >::U().

6.1.3.10 template<int N, class T0> template<class TypeS> bool esso::Array< N, T0 >::Refer (const Array< N, TypeS > & *src*) `[inline]`

Reinitialize this object as an identical copy of `src` (refering to the same data area as `src`)

Returns:

true on success or false if operation failed

References esso::Array< N, T0 >::Data(), esso::Array< N, T0 >::L(), esso::Array< N, T0 >::Origo(), esso::Array< N, T0 >::Stride(), and esso::Array< N, T0 >::U().

6.1.3.11 template<int N, class T0> bool esso::Array< N, T0 >::Clone () `[inline]`

Same as Clone(*this).

Returns:

true on success or false if operation failed

Referenced by esso::Clone().

6.1.3.12 template<int N, class T0> template<class TypeS> bool esso::Array< N, T0 >::Clone (const Array< N, TypeS > & *src*) `[inline]`

Release this object from it's element data and allocate a new data set with index bounds identical to those of `src`. Finally copy the element data from `src`

Note that this is the same as Realloc followed by element-wise copy.

Returns:

true on success or false if operation failed

References esso::Array< N, T0 >::IsNull(), and esso::Array< N, T0 >::Realloc().

6.1.3.13 template<int N, class T0> template<class TypeS> Array< N, T0 > & esso::Array< N, T0 >::Extend (const Array< N-1, TypeS > & *src*, IXTYPE *l0*, IXTYPE *u0*) `[inline]`

Create a reference to `src` by setting the stride for dimension 0 to 0, that is, all elements along dimension 0 refer to the same element position. New index bounds for dimension zero are given as arguments.

References esso::Array< N, T0 >::Data(), esso::Array< N, T0 >::L(), esso::Array< N, T0 >::Origo(), esso::Array< N, T0 >::Stride(), and esso::Array< N, T0 >::U().

6.1.3.14 template<int N, class T0> template<class TypeS> Array< N, T0 > & esso::Array< N, T0 >::Rmdim (const Array< N+1, TypeS > & *src*, DIMT *n*, IXTYPE *i*) `[inline]`

Create a reference to a slice of `src` by cutting along dimension `n` at index `i`

References esso::Array< N, T0 >::Data(), esso::Array< N, T0 >::L(), esso::Array< N, T0 >::Origo(), esso::Array< N, T0 >::Stride(), and esso::Array< N, T0 >::U().

6.1.3.15 template<int N, class T0> template<class TypeS> Array& esso::Array< N, T0 >::operator= (const Array< N, TypeS > & *src*) `[inline]`

Create an identical reference to `src` data if possible (same as Refer(src)), otherwise create a new data area and copy `src` elements into it (same as Clone(src))

6.1.3.16 template<int N, class T0> Array& esso::Array< N, T0 >::operator= (const Array< N, T0 > & *src*) `[inline]`

Create an identical reference to `src` data (same as Refer(src))

6.1.3.17 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::Shift (DIMT *n*, IXTYPE *i*) `[inline]`

Shift data and valid index range along dimension `n`

Referenced by esso::Array< N, T0 >::Shift().

6.1.3.18 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::Shift (const Ix< N > & *ix*) `[inline]`

Shift data and valid index range along all dimensions

References esso::Array< N, T0 >::Shift().

6.1.3.19 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::Permute (DIMT *n1*, DIMT *n2*) `[inline]`

Permute dimensions by swapping two

6.1.3.20 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::Permute (const Permutation< N > & *nx*) `[inline]`

Permute dimensions

6.1.3.21 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::Restrict (DIMT *n*, IXTYPE *l0*, IXTYPE *u0*) `[inline]`

Restrict valid index range along dimension `n`

Resulting index range along `n` is the intersection of the given range [l0,u0[and the previous index range [L(n),U(n)[

6.1.3.22 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::Restrict (const Ix< N > & *l0*, const Ix< N > & *u0*) `[inline]`

Restrict valid index range along all dimensions

6.1.3.23 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::RestrictL (DIMT *n*, IXTYPE *l0*) [inline]

Restrict lower index bound along dimension `n`

6.1.3.24 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::RestrictL (const Ix< N > & *l0*) [inline]

Restrict lower index bound along all dimensions

6.1.3.25 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::RestrictU (DIMT *n*, IXTYPE *u0*) [inline]

Restrict upper index bound along dimension `n`

6.1.3.26 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::RestrictU (const Ix< N > & *u0*) [inline]

Restrict upper index bound along all dimensions

6.1.3.27 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::Flip (DIMT *n*) [inline]

Reverse order of elements along dimension `n` and adjust valid index range so that element at (0,0,...,0) is unchanged

6.1.3.28 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::Flip (const Ix< N, bool > & *r*) [inline]

Reverse order of elements along all dimensions and adjust valid index range so that element at (0,0,...,0) is unchanged

6.1.3.29 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::Reverse (DIMT *n*) [inline]

Reverse order of elements along dimension `n`. Valid index range is unchanged

6.1.3.30 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::Reverse (const Ix< N, bool > & *r*) [inline]

Reverse order of elements along all dimensions, Valid index range is unchanged

6.1.3.31 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::Restride (DIMT *n*, IXTYPE *s*) [inline]

Set this object to point to every `s` 'th element along dimension `n`

Referenced by esso::Array< N, T0 >::Restride().

6.1.3.32 template<int N, class T0> Array< N, T0 > & esso::Array< N, T0 >::Restride (const Ix< N > & *s*) [inline]

Set this object to point to every `s`[n] 'th element along dimension `n` for all `n` (along all dimensions)

References esso::Array< N, T0 >::Restride().

6.1.3.33 template<int N, class T0> T& esso::Array< N, T0 >::operator() (IXTYPE *i0*) const [inline]

Get 1-dimensional array element at given position

6.1.3.34 template<int N, class T0> T& esso::Array< N, T0 >::operator() (IXTYPE *i0*, IXTYPE *i1*) const [inline]

Get 2-dimensional array element at given position

6.1.3.35 template<int N, class T0> T& esso::Array< N, T0 >::operator() (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*) const [inline]

Get 3-dimensional array element at given position

6.1.3.36 template<int N, class T0> T& esso::Array< N, T0 >::operator() (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*, IXTYPE *i3*) const [inline]

Get 4-dimensional array element at given position

6.1.3.37 template<int N, class T0> T& esso::Array< N, T0 >::operator() (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*, IXTYPE *i3*, IXTYPE *i4*) const [inline]

Get 5-dimensional array element at given position

6.1.3.38 template<int N, class T0> T& esso::Array< N, T0 >::operator() (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*, IXTYPE *i3*, IXTYPE *i4*, IXTYPE *i5*) const [inline]

Get 6-dimensional array element at given position

6.1.3.39]

template<int N, class T0> T& **esso::Array**< N, T0 >::operator[] (const **Ix**< N > & *ix*) const [inline]

Get element at given position

Referenced by esso::Array< N, Type >::Start(), and esso::Array< N, Fix< T0 > >::Start().

The documentation for this class was generated from the following file:

- flex.h

6.2 esso::Array< N, Fix< T0 > > Class Template Reference

Multidimensional, efficient array class for large data sets.

`#include <fix.h>`

Public Member Functions

- T & **operator()** (**IXTYPE** i0) const
- T & **operator()** (**IXTYPE** i0, **IXTYPE** i1) const
- T & **operator()** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2) const
- T & **operator()** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2, **IXTYPE** i3) const
- T & **operator()** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2, **IXTYPE** i3, **IXTYPE** i4) const
- T & **operator()** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2, **IXTYPE** i3, **IXTYPE** i4, **IXTYPE** i5) const
- T & **operator[]** (const **Ix**< N > &ix) const

constructor

- **Array** ()
- **Array** (const **Ix**< varrank > &len0)
- **Array** (const **Ix**< varrank > &len0, const **Ix**< N > &l0)
- **Array** (const **Array** &src)
- template<class TypeS>
  Array (const **Array**< N, TypeS > &src)
- **Array** (**IXTYPE** i0)
- **Array** (**IXTYPE** i0, **IXTYPE** i1)
- **Array** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2)
- **Array** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2, **IXTYPE** i3)
- **Array** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2, **IXTYPE** i3, **IXTYPE** i4)
- **Array** (**IXTYPE** i0, **IXTYPE** i1, **IXTYPE** i2, **IXTYPE** i3, **IXTYPE** i4, **IXTYPE** i5)

const

- **IXTYPE L** (**DIMT** n) const
- **IXTYPE U** (**DIMT** n) const
- **IXTYPE Len** (**DIMT** n) const
- **IXTYPE Stride** (**DIMT** n) const
- T * **Origo** () const
- T * **Start** () const
- const MultiPtr< T > & **Data** () const
- bool **IsNull** () const

nonconst

- bool **Dealloc** ()
- bool **Realloc** (const **Ix**< varrank > &len0)
- bool **Realloc** (const **Ix**< varrank > &len0, const **Ix**< N > &l0)
- bool **Realloc** (const **Ix**< varrank > &len0, const **Ix**< N > &l0, const MultiPtr< T > &data0, T *origo0)
- template<class TypeS>
  bool **Realloc** (const **Array**< N, TypeS > &src)

- template<class TypeS>
  bool **Refer** (const **Array**< N, TypeS > &src)
- bool **Clone** ()
- template<class TypeS>
  bool **Clone** (const **Array**< N, TypeS > &src)
- template<class TypeS>
  Array & **operator=** (const **Array**< N, TypeS > &src)
- **Array** & **operator=** (const **Array** &src)
- **Array** & **Shift** (**DIMT** n, **IXTYPE** i)
- **Array** & **Shift** (const **Ix**< N > &ix)
- **Array** & **Restrict** (**DIMT** n, **IXTYPE** l0, **IXTYPE** u0)
- **Array** & **Restrict** (const **Ix**< N > &l0, const **Ix**< N > &u0)
- **Array** & **RestrictL** (**DIMT** n, **IXTYPE** l0)
- **Array** & **RestrictL** (const **Ix**< N > &l0)
- **Array** & **RestrictU** (**DIMT** n, **IXTYPE** u0)
- **Array** & **RestrictU** (const **Ix**< N > &u0)

6.2.1 Detailed Description

template<int N, class T0> class esso::Array< N, Fix< T0 > >

Multidimensional, efficient array class for large data sets.

Use this array for efficient numerical computations or other applications where powerful and efficient arrays are needed for large data sets.

This specialization of Array<N,T> is often more efficient than the default **Array** (p. 41) type.

The difference between Fix and standard **Array** (p. 41) is:

- A Fix **Array** (p. 41) cannot permute its' ranks at runtime, the storage order of ranks in memory is fixed.
- A Fix **Array** (p. 41) is in general faster than a flex array, because it's lowest dimension always has stride 1.
- A Fix **Array** (p. 41) can have a fixed length of one or more of it's lowest ranks, a standard **Array** (p. 41) cannot.
- A standard **Array** (p. 41) can refer to any slice of a larger array, Fix Arrays can (in general) only refer to slices of other Fix arrays, and then only to the lower ranks.

As an example of fixing lower index bounds at compile time, **Array** (p. 41)<3,Fix<int[5]> > declares a three-dimensional array whose lowest dimension is always of length 5. Or even **Array** (p. 41)<3,Fix<int[5][10]> > where the two lowest ranges are given as template arguments.

There may be two reasons for doing this. First it increases type safety, if the lowest dimension range is known then the compiler should be told about it. Second it may increase the compiler's ability to optimize some loops.

To retrieve a fortran-compatible pointer to first element, use member function Start.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 template<int N, class T0> esso::Array< N, Fix< T0 > >::Array () `[inline]`

Default constructor. Initialize array with no data and index bounds zero

6.2.2.2 template<int N, class T0> esso::Array< N, Fix< T0 > >::Array (const Ix< varrank > & *len0*) [inline]

Constructor with given number of elements and lower index bound = 0

6.2.2.3 template<int N, class T0> esso::Array< N, Fix< T0 > >::Array (const Ix< varrank > & *len0*, const Ix< N > & *l0*) [inline]

Constructor with given number of elements and given lower index bound

6.2.2.4 template<int N, class T0> esso::Array< N, Fix< T0 > >::Array (const Array< N, Fix< T0 > > & *src*) [inline]

Copy constructor Creates a reference to the same memory area as *src*

6.2.2.5 template<int N, class T0> template<class TypeS> esso::Array< N, Fix< T0 > >::Array (const Array< N, TypeS > & *src*) [inline]

Copy constructor from another array type object If possible, create a reference to the same memory area as *src*. If this is not possible, create a new memory area and copy *src* 's elements

6.2.2.6 template<int N, class T0> esso::Array< N, Fix< T0 > >::Array (IXTYPE *i0*) [inline]

1-dimensional array of given length and lower bounds =0 (only compiles for Array<1,T>)

6.2.2.7 template<int N, class T0> esso::Array< N, Fix< T0 > >::Array (IXTYPE *i0*, IXTYPE *i1*) [inline]

2-dimensional array of given length and lower bounds =0 (only compiles for Array<2,T>)

6.2.2.8 template<int N, class T0> esso::Array< N, Fix< T0 > >::Array (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*) [inline]

3-dimensional array of given length and lower bounds =0 (only compiles for Array<3,T>)

6.2.2.9 template<int N, class T0> esso::Array< N, Fix< T0 > >::Array (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*, IXTYPE *i3*) [inline]

4-dimensional array of given length and lower bounds =0 (only compiles for Array<4,T>)

6.2.2.10 template<int N, class T0> esso::Array< N, Fix< T0 > >::Array (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*, IXTYPE *i3*, IXTYPE *i4*) [inline]

5-dimensional array of given length and lower bounds =0 (only compiles for Array<5,T>)

6.2.2.11 template<int N, class T0> esso::Array< N, Fix< T0 > >::Array (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*, IXTYPE *i3*, IXTYPE *i4*, IXTYPE *i5*) [inline]

6-dimensional array of given length and lower bounds =0 (only compiles for Array<6,T>)

6.2.3 Member Function Documentation

6.2.3.1 template<int N, class T0> IXTYPE esso::Array< N, Fix< T0 > >::L (DIMT *n*) const [inline]

Lower index bounds L(n) is the lowest index allowed on dimension n

6.2.3.2 template<int N, class T0> IXTYPE esso::Array< N, Fix< T0 > >::U (DIMT *n*) const [inline]

Upper index bounds U(n) is one more than the largest index allowed on dimension n

6.2.3.3 template<int N, class T0> IXTYPE esso::Array< N, Fix< T0 > >::Len (DIMT *n*) const [inline]

Convenience function giving the number of array elements along given dimension. Simply put, Len(n) is U(n) - L(n)

6.2.3.4 template<int N, class T0> T* esso::Array< N, Fix< T0 > >::Origo () const [inline]

Pointer to element at index (0,0,...,0) Useful in some rare circumstances. Note that this pointer does not point to a valid element if (0,0,...,0) is outside valid index range.

6.2.3.5 template<int N, class T0> T* esso::Array< N, Fix< T0 > >::Start () const [inline]

Pointer to first valid element i.e. element at lowest index bound

Use this function to get pointer to pass to fortran functions.

References esso::Array< N, T0 >::operator[]().

6.2.3.6 template<int N, class T0> bool esso::Array< N, Fix< T0 > >::IsNull () const [inline]

Returns:

true if this array has no data allocated

6.2.3.7 template<int N, class T0> bool esso::Array< N, Fix< T0 > >::Dealloc () [inline]

Release this object from it's element data and set index bounds to zero

Returns:

always returns true

6.2.3.8 template<int N, class T0> bool esso::Array< N, Fix< T0 > >::Realloc (const Ix< varrank > & *len0*) [inline]

Release this object from it's element data and allocate a new data set with given size (lower index bounds zero)

Returns:

always returns true

6.2.3.9 template<int N, class T0> bool esso::Array< N, Fix< T0 > >::Realloc (const Ix< varrank > & *len0*, const Ix< N > & *l0*) [inline]

Release this object from it's element data and allocate a new data set with given size and given lower index bounds

Returns:

always returns true

6.2.3.10 template<int N, class T0> template<class TypeS> bool esso::Array< N, Fix< T0 > >::Realloc (const Array< N, TypeS > & *src*) [inline]

Release this object from it's element data and allocate a new, uninitialized data set with index bounds identical to those of *src*.

Returns:

true if successful or false if operation failed

6.2.3.11 template<int N, class T0> template<class TypeS> bool esso::Array< N, Fix< T0 > >::Refer (const Array< N, TypeS > & *src*) [inline]

Reinitialize this object as an identical copy of *src* (refering to the same element data as *src*)

Returns:

true if successful or false if operation failed

6.2.3.12 template<int N, class T0> bool esso::Array< N, Fix< T0 > >::Clone () [inline]

Same as Clone(∗this).

Returns:

always returns true

References esso::Clone().

6.2.3.13 template<int N, class T0> template<class TypeS> bool esso::Array< N, Fix< T0 > >::Clone (const Array< N, TypeS > & *src*) `[inline]`

Release this object from it's element data and allocate a new data set with index bounds identical to those of *src*. Finally copy the element data from *src*

Note that this is the same as Realloc followed by element-wise copy.

Returns:

true if successful or false if operation failed

References esso::Array< N, T0 >::IsNull(), and esso::Array< N, T0 >::Realloc().

6.2.3.14 template<int N, class T0> template<class TypeS> Array& esso::Array< N, Fix< T0 > >::operator= (const Array< N, TypeS > & *src*) `[inline]`

Create an identical reference to *src* data if possible (same as Refer(src)), otherwise create a new data area and copy *src* 's elements into it (same as Clone(src))

6.2.3.15 template<int N, class T0> Array& esso::Array< N, Fix< T0 > >::operator= (const Array< N, Fix< T0 > > & *src*) `[inline]`

Create an identical reference to *src* data (same as Refer(src))

References esso::Array< N, T0 >::data, esso::Array< N, T0 >::l, esso::Array< N, T0 >::origo, esso::Array< N, T0 >::stride, and esso::Array< N, T0 >::u.

6.2.3.16 template<int N, class T0> Array< N, Fix< T0 > > & esso::Array< N, Fix< T0 > >::Shift (DIMT *n*, IXTYPE *i*) `[inline]`

Shift data and valid index range along dimension n

6.2.3.17 template<int N, class T0> Array< N, Fix< T0 > > & esso::Array< N, Fix< T0 > >::Shift (const Ix< N > & *ix*) `[inline]`

Shift data and valid index range along all dimensions

References esso::Shift().

6.2.3.18 template<int N, class T0> Array< N, Fix< T0 > > & esso::Array< N, Fix< T0 > >::Restrict (DIMT *n*, IXTYPE *l0*, IXTYPE *u0*) `[inline]`

Restrict valid index range along dimension n . Resulting index range along n is intersection of [l0,u0[and [L(n),U(n)[.

6.2.3.19 template<int N, class T0> Array< N, Fix< T0 > > & esso::Array< N, Fix< T0 > >::Restrict (const Ix< N > & *l0*, const Ix< N > & *u0*) `[inline]`

Restrict valid index range along all dimensions

6.2.3.20 template<int N, class T0> Array< N, Fix< T0 > > & esso::Array< N, Fix< T0 > >::RestrictL (DIMT *n*, IXTYPE *l0*) `[inline]`

Restrict lower index bound along dimension n

6.2.3.21 template<int N, class T0> Array< N, Fix< T0 > > & esso::Array< N, Fix< T0 > >::RestrictL (const Ix< N > & *l0*) `[inline]`

Restrict lower index bound along all dimensions

6.2.3.22 template<int N, class T0> Array< N, Fix< T0 > > & esso::Array< N, Fix< T0 > >::RestrictU (DIMT *n*, IXTYPE *u0*) `[inline]`

Restrict upper index bound along dimension n

6.2.3.23 template<int N, class T0> Array< N, Fix< T0 > > & esso::Array< N, Fix< T0 > >::RestrictU (const Ix< N > & *u0*) `[inline]`

Restrict upper index bound along all dimensions

6.2.3.24 template<int N, class T0> T& esso::Array< N, Fix< T0 > >::operator() (IXTYPE *i0*) const `[inline]`

Get 1-dimensional array element at given position

6.2.3.25 template<int N, class T0> T& esso::Array< N, Fix< T0 > >::operator() (IXTYPE *i0*, IXTYPE *i1*) const `[inline]`

Get 2-dimensional array element at given position

6.2.3.26 template<int N, class T0> T& esso::Array< N, Fix< T0 > >::operator() (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*) const `[inline]`

Get 3-dimensional array element at given position

6.2.3.27 template<int N, class T0> T& esso::Array< N, Fix< T0 > >::operator() (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*, IXTYPE *i3*) const `[inline]`

Get 4-dimensional array element at given position

6.2.3.28 template<int N, class T0> T& esso::Array< N, Fix< T0 > >::operator() (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*, IXTYPE *i3*, IXTYPE *i4*) const `[inline]`

Get 5-dimensional array element at given position

6.2.3.29 template<int N, class T0> T& esso::Array< N, Fix< T0 > >::operator() (IXTYPE *i0*, IXTYPE *i1*, IXTYPE *i2*, IXTYPE *i3*, IXTYPE *i4*, IXTYPE *i5*) const [inline]

Get 6-dimensional array element at given position

6.2.3.30]

template<int N, class T0> T& **esso::Array**< N, Fix< T0 > >::operator[] (const **Ix**< N > & *ix*) const `[inline]`

Get element at given position

The documentation for this class was generated from the following file:

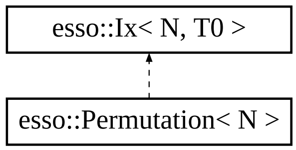
- fix.h

6.3 esso::Ix< N, T0 > Class Template Reference

Represents a small, fast vector.

`#include <ix.h>`

Inheritance diagram for esso::Ix< N, T0 >::



Public Member Functions

construction

- **Ix** ()
- **Ix** (T i0)
- **Ix** (const **Ix** &i0)

construction from sequence of elements

- **Ix** (T0 i0, T0 i1)
- **Ix** (T0 i0, T0 i1, T0 i2)
- **Ix** (T0 i0, T0 i1, T0 i2, T0 i3)
- **Ix** (T0 i0, T0 i1, T0 i2, T0 i3, T0 i4)
- **Ix** (T0 i0, T0 i1, T0 i2, T0 i3, T0 i4, T0 i5)

const

- T0 **operator[ ]** (**DIMT** n) const
- bool **operator==** (const T &src) const
- bool **operator==** (const **Ix** &src) const
- bool **operator!=** (const T &src) const
- bool **operator!=** (const **Ix** &src) const
- template<DIMT LEN>
  Ix< LEN, T0 > **Sub** (**DIMT** start=0) const

nonconst

- T0 & **operator[ ]** (**DIMT** n)
- **Ix** & **operator=** (T src)
- template<class TT>
  Ix & **operator+=** (const **Ix**< N, TT > &src)
- **Ix** & **operator+=** (const T &src)
- template<class TT>
  Ix & **operator-=** (const **Ix**< N, TT > &src)
- **Ix** & **operator-=** (const T &src)
- **Ix** & **operator*=** (const T &src)
- **Ix** & **operator/=** (const T &src)
- template<int N1>
  Ix & **Concat** (const **Ix**< N1, T0 > &p1, const **Ix**< N-N1, T0 > &p2)
- template<int M>
  Ix< N+M, T0 > **operator|** (const **Ix**< M, T0 > &p2) const

6.3.1 Detailed Description

template<int N, class T0> class esso::Ix< N, T0 >

Represents a small, fast vector.

Use this object for small vectors where the length of the vector is known at compile time. (It is possible to use it for larger datasets too, but it should be noted that all data is allocated on the stack, no dynamic memory is allocated by these objects).

6.3.2 Constructor & Destructor Documentation

6.3.2.1 template<int N, class T0> esso::Ix< N, T0 >::Ix () `[inline]`

Default constructor. Just creates the vector without initializing the elements

6.3.2.2 template<int N, class T0> esso::Ix< N, T0 >::Ix (T *i0*) `[inline, explicit]`

Constructor which inializes all elements with scalar (copied into all elements).

6.3.2.3 template<int N, class T0> esso::Ix< N, T0 >::Ix (const Ix< N, T0 > & *i0*) `[inline]`

Copy constructor. Copies elementwise

6.3.2.4 template<int N, class T0> esso::Ix< N, T0 >::Ix (T0 *i0*, T0 *i1*) `[inline]`

Constructor from two integers (only compiles for Ix<2,T>)

6.3.2.5 template<int N, class T0> esso::Ix< N, T0 >::Ix (T0 *i0*, T0 *i1*, T0 *i2*) `[inline]`

Constructor from three integers (only compiles for Ix<3,T>)

6.3.2.6 template<int N, class T0> esso::Ix< N, T0 >::Ix (T0 *i0*, T0 *i1*, T0 *i2*, T0 *i3*) `[inline]`

Constructor from four integers (only compiles for Ix<4,T>)

6.3.2.7 template<int N, class T0> esso::Ix< N, T0 >::Ix (T0 *i0*, T0 *i1*, T0 *i2*, T0 *i3*, T0 *i4*) `[inline]`

Constructor from five integers (only compiles for Ix<5,T>)

6.3.2.8 template<int N, class T0> esso::Ix< N, T0 >::Ix (T0 *i0*, T0 *i1*, T0 *i2*, T0 *i3*, T0 *i4*, T0 *i5*) `[inline]`

Constructor from six integers (only compiles for Ix<6,T>)

6.3.3 Member Function Documentation

6.3.3.1]

template<int N, class T0> T0 **esso::Ix**< N, T0 >::operator[] (**DIMT** *n*) const `[inline]`

Elementwise access

Reimplemented in **esso::Permutation**< **N** > (p. 64).

6.3.3.2 template<int N, class T0> bool esso::Ix< N, T0 >::operator== (const T & *src*) const `[inline]`

Returns:

true if this object is elementwise equal to *src*

6.3.3.3 template<int N, class T0> bool esso::Ix< N, T0 >::operator== (const Ix< N, T0 > & *src*) const `[inline]`

Returns:

true if this object is elementwise equal to *src*

6.3.3.4 template<int N, class T0> bool esso::Ix< N, T0 >::operator!= (const T & *src*) const `[inline]`

Returns:

true if this object is not elementwise equal to *src*

6.3.3.5 template<int N, class T0> bool esso::Ix< N, T0 >::operator!= (const Ix< N, T0 > & *src*) const `[inline]`

Returns:

true if this object is not elementwise equal to *src*

6.3.3.6 template<int N, class T0> template<DIMT LEN> Ix<LEN,T0> esso::Ix< N, T0 >::Sub (DIMT *start* = `0`) const `[inline]`

Returns:

an `Ix` (p. 58) object whose elements are a sub-range of the elements of this `Ix` (p. 58) object

6.3.3.7]

template<int N, class T0> T0& **esso::Ix**< N, T0 >::operator[] (**DIMT** *n*) `[inline]`

Elementwise access

6.3.3.8 template<int N, class T0> Ix& esso::Ix< N, T0 >::operator= (T *src*) [inline]

Assignment from scalar (value is copied into all elements)

6.3.3.9 template<int N, class T0> template<class TT> Ix& esso::Ix< N, T0 >::operator+= (const Ix< N, TT > & *src*) [inline]

Addition

6.3.3.10 template<int N, class T0> Ix& esso::Ix< N, T0 >::operator+= (const T & *src*) [inline]

Addition with scalar

6.3.3.11 template<int N, class T0> template<class TT> Ix& esso::Ix< N, T0 >::operator-= (const Ix< N, TT > & *src*) [inline]

Subtraction

6.3.3.12 template<int N, class T0> Ix& esso::Ix< N, T0 >::operator-= (const T & *src*) [inline]

Subtraction with scalar

6.3.3.13 template<int N, class T0> Ix& esso::Ix< N, T0 >::operator*= (const T & *src*) [inline]

Multiplication with scalar

6.3.3.14 template<int N, class T0> Ix& esso::Ix< N, T0 >::operator/= (const T & *src*) [inline]

Quotient with scalar

6.3.3.15 template<int N, class T0> template<int N1> Ix& esso::Ix< N, T0 >::Concat (const Ix< N1, T0 > & *p1*, const Ix< N-N1, T0 > & *p2*) [inline]

Concatenate two vectors i.e. if v1 has dim n1 and v2 has dim n2 then Concat(v1,v2)[n] == v1[n] if n<n1 and v2[n-n1] otherwise

Returns:

reference to this object

Referenced by esso::Ix< N, short >::operator|().

6.3.3.16 template<int N, class T0> template<int M> Ix<N+M,T0> esso::Ix< N, T0 >::operator| (const Ix< M, T0 > & *p2*) const `[inline]`

Returns:

Concatenation of this two **Ix** (p. 58) objects

See also:

Ix::Concat (p. 61)

The documentation for this class was generated from the following file:

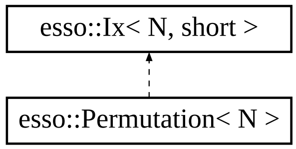
- ix.h

6.4 esso::Permutation< N > Class Template Reference

Represents a permutation of dimensions.

`#include <permutation.h>`

Inheritance diagram for esso::Permutation< N >::



Public Member Functions

construction

- **Permutation** ()
- **Permutation** (int r)

const

- **DIMT operator[ ]** (**DIMT** n) const
- bool **operator==** (const **Permutation**< N > &src) const
- bool **operator!=** (const **Permutation**< N > &src) const
- template<class TT>
  Ix< N, TT > **operator∗** (const **Ix**< N, TT > &i) const
- **Permutation**< N > **operator∗** (const **Permutation**< N > &p) const
- **Permutation**< N > **operator[ ]** (const **Permutation**< N > &p) const
- bool **IsEven** () const

nonconst

- bool **Setup** (const **Ix**< N, **DIMT** > &src)
- bool **SetupSwap** (**DIMT** n1, **DIMT** n2)
- **Permutation**< N > & **Invert** ()
- **Permutation**< N > **operator∼** () const
- template<int N1>
  Permutation< N > & **Concat** (const **Permutation**< N1 > &p1, const **Permutation**< N-N1 > &p2)
- template<int M>
  Permutation< N+M > **operator|** (const **Permutation**< M > &p2) const

6.4.1 Detailed Description

template<int N> class esso::Permutation< N >

Represents a permutation of dimensions.

If n is a dimension, then the transformed dimension is p[n] where p is a **Permutation** (p. 63) object.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 template<int N> esso::Permutation< N >::Permutation () [inline]

Default constructor. Creates unity permutation.

Referenced by esso::Permutation< N >::operator~().

6.4.2.2 template<int N> esso::Permutation< N >::Permutation (int *r*) [inline, explicit]

Create a permutation that is a cyclic rotation of dimensions, that is p[n] == (n+r) modulo N. Negative values for *r* are allowed

6.4.3 Member Function Documentation

6.4.3.1]

template<int N> **DIMT esso::Permutation**< N >::operator[] (**DIMT** *n*) const [inline]

Returns:

permuted dimension for dimension *n*

Reimplemented from **esso::Ix**< **N, T0** > (p. 60).

6.4.3.2 template<int N> bool esso::Permutation< N >::operator== (const Permutation< N > & *src*) const [inline]

Returns:

true iff this object is equal to *src*

6.4.3.3 template<int N> bool esso::Permutation< N >::operator!= (const Permutation< N > & *src*) const [inline]

Returns:

true iff this object is not equal to *src*

6.4.3.4 template<int N> template<class TT> Ix<N,TT> esso::Permutation< N >::operator* (const Ix< N, TT > & *i*) const [inline]

Transform an **Ix** (p. 58) object by this permutation. I.e. if n is a dimension then (p*i)[p[n]] = i[n]

Note that this works just like matrix-vector multiplication where the **Permutation** (p. 63) object acts like a permutation matrix with respect to multiplication.

Returns:

vector *i* transformed into permuted space

6.4.3.5 template<int N> Permutation<N> esso::Permutation< N >::operator* (const Permutation< N > & *p*) const `[inline]`

Compose a **Permutation** (p. 63) from two objects. I.e. if if p1 and p2 are permutations and n is a dimension then (p1*p2)[n] == p1[p2[n]] Note that this works just like matrix-matrix multiplication where the **Permutation** (p. 63) object acts like a permutation matrix with respect to multiplication.

Returns:

product of two permutations i.e.

References esso::Ix< N, T0 >::d.

6.4.3.6]

template<int N> **Permutation**<N> **esso::Permutation**< N >::operator[] (const **Permutation**< N > & *p*) const `[inline]`

Transform a permutation by another permutation. I.e. if p1 and p2 are permutations and n is a dimension then (p1[p2])[n] == p1[p2[~p1[n]]]

Note that this means that p1[p2] is equivalent to p1*p2*~p1.

Returns:

permutation *p* transformed into permuted space i.e.

References esso::Ix< N, T0 >::d.

6.4.3.7 template<int N> bool esso::Permutation< N >::IsEven () const `[inline]`

Find out if this is an 'even' permutation, that is if it can be composed by an even number of 'elementary permutations'. An elementary permutation in this context is a permutation of two consecutive dimensions e.g. (1,2)->(2,1).

Knowing the 'even' property of a permutation is important in many applications.

Returns:

true iff this object is an 'even' permutation.

References esso::Ix< N, T0 >::d.

6.4.3.8 template<int N> bool esso::Permutation< N >::Setup (const Ix< N, DIMT > & *src*) `[inline]`

Create object.and checks so that arguments are ok. On failure the contents of this object is unchanged.

Returns:

true iff successful

6.4.3.9 template<int N> bool esso::Permutation< N >::SetupSwap (DIMT *n1*, DIMT *n2*) `[inline]`

Create object as single permutation of two dimensions. Checks so that arguments are ok. On failure this object is unchanged.

Returns:

true iff successful

6.4.3.10 template<int N> Permutation<N>& esso::Permutation< N >::Invert () `[inline]`

Set this object to it's own inverse

Returns:

reference to this object

References esso::Ix< N, T0 >::d.

6.4.3.11 template<int N> Permutation<N> esso::Permutation< N >::operator~ () const `[inline]`

Returns:

A permutation which is the inverse of this permutation.

See also:

Permutation::Inverse

References esso::Permutation< N >::Permutation().

6.4.3.12 template<int N> template<int N1> Permutation<N>& esso::Permutation< N >::Concat (const Permutation< N1 > & *p1*, const Permutation< N-N1 > & *p2*) `[inline]`

Concatenate two permutations. if p1 has dim n1 and p2 has dim n2 then (p1|p2)[n] == p1[n] if n<n1 and p2[n-n1]+n1 otherwise

Returns:

new permutation object

Referenced by esso::Permutation< N >::operator|().

6.4.3.13 template<int N> template<int M> Permutation<N+M> esso::Permutation< N >::operator| (const Permutation< M > & *p2*) const `[inline]`

Returns:

A permutation which is the concatenation of this object with *p2*

See also:

Permutation::Concat (p. 66)

References esso::Permutation< N >::Concat().

The documentation for this class was generated from the following file:

- permutation.h

Index