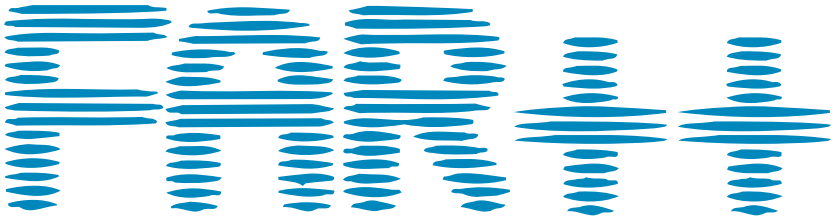


# Fortran Arrays in C++

C. S. Brady, H. Ratcliffe

# Contents

## FAR++ - Fortran arrays in C++

> …the determined Real Programmer can write FORTRAN programs in any language.

Ed Post, “Real Programers Don’t use Pascal”

### Aims and purpose

While modern Fortran is perhaps the best overall programming language for scientific computation there are more and more circumstances where scientific software is developed in C++. C++ is a very good language for structured, object oriented and generic programming but it is not actually terribly well suited to scientific programming. In particular arrays in Fortran are much better than their C++ equivalents and since scientists are often not professional programmers Fortran is often easier for them to learn and use. This is particularly troublesome because C++ is often used to teach undergraduate programming courses based on it being a more transferable skill, but usually stops short of teaching the skills that are needed to be an effective developer of scientific software.

The aim of this library is to move the general Fortran approach to arrays and the Fortran array function library to C++ and have it still be comparable in speed to both native C++ and Fortran. It allows you to create rank N arrays (where N is entirely arbitrary) of any type that is both default constructible and either copy assignable or copy constructible (other constructors and assignment operators will be used if present but copy assignment or copy construction are needed)

AT THE MOMENT THIS LIBRARY SHOULD BE CONSIDERED BETA SOFTWARE. Some features that are in the public release can only be considered of alpha quality. It should not be used for any mission critical purposes and no warranty or underaking is provided as to its suitability for any purpose. Use it entirely at your own risk.

The developers recommend this library for

- Teaching purposes - it is much easier to write scientific software using arrays with FAR++ than with native C++
- Rapid development of prototypes - Fortran is often much easier than C++ to quickly write a proof of concept code, this library makes development in C++ much quicker

This library might be usable for

- Small to medium scale simulations - This is a qualified “should”. The library has been tested and doesn’t show any strange performance problems, incorrect answers or memory leaks in testing. That doesn’t mean that there aren’t any if you try hard enough. If care is taken to test your code it should be usable. Future releases may well be well suited to this type of problem

- You are moving a Fortran code to C++ - It might make the transfer a bit easier, but you do have to be careful about replicating pathologies of the original design

This library probably shouldn't be used for

- Large scale simulations - This just hasn't been tested enough in anger to be let loose on simulations that can use a lot of power and cost a lot of money. Obviously, your choice, but the authors wouldn't recommend it!

Finally, if you are looking at this library as a way of making a code of yours easier to write in C++, we would *strongly recommend* looking first at whether you actually need the features that C++ provides rather than the features that Fortran provides. If you don't then it might be worth considering switching to Fortran for your project. Fortran is a thoroughly modern language with many features that make it a good choice for scientific software development. This library aims to smooth over some of the rough edges of C++ when used for this task, but there is no substitute for a tool specifically designed for your problem

## Notes on capitalisation

Fortan is completely case insensitive, but C++ is case sensitive. All functions in FAR++ either match their Fortran equivalents in lower case or, in cases where Fortran uses optional parameters and keyword calling in a way that C++ cannot replicate, the lower case Fortran names are supplemented with suffixes to select between versions of the function. Objects in C++ are almost all named in camel case (lower case first letter then other words capitalised and no spaces between words, so camel case itself would become camelCase). The exception is the core `Array` class which is named with a leading capital A to reduce the risk of collisions with `std::array`, and the specifically ordered `FortranArray` and `CArray` classes where the name includes proper nouns.

## Using FAR++

### Boilerplate

A few technical or conventional terms are used in this document for the sake of precision. They are

- rvalue - A "right value" - something that ONLY exists on the right hand side of an assignment, for example something returned by a function or created by using operators will be an rvalue. Technically there is more than one kind of rvalue in C++ but they are all considered equivalent in FAR++

- contiguous - Data that exists one piece immediately after the other in the computers memory. FAR++ arrays are created contiguous, but a user can create non-contiguous arrays by slicing out parts of an array. There are places where FAR++ can operate faster and/or with lower memory overhead on contiguous arrays
- core type and array type - FAR++ arrays are themselves types, so in order to avoid ambiguity, we describe the whole array to be an `array type` and the type of all of the member elements of the array as being the `core type` of the array

FAR++ is a header only library. To make use of it simply include the header "far.h" and make sure that the include directory from this repository is included in the list of include directories. All of the functions and classes are in the namespace "far". It should be safe to use this namespace for a more Fortran-like experience

FAR++ uses C++20 features so your compiler must support C++ 20. Some compilers, notably both g++ and clang++ need you to manually tell them that you want to use the C++20 standard. You can do this for both of them using "–std=c++20" on the compile line when compiling code. If you see a lot of errors when compiling a FAR++ program, check that you are compiling in C++ 20 mode before attempting any other troubleshooting.

FAR++ is a header only library so it relies on the inliner for good performance. Unlike actual Fortran it is difficult to use FAR++ without optimisation. Performance can be literally hundreds of times slower on `O0` compared to `O3` and even `O1` can be 10 times slower. Partly this is the reliance on inlining and partly it is just that Fortran array functions and operators are generally implemented as calls to library code that is still compiled with optimisation even if your code is not.

Link time optimisation can massively improve performance for FAR++ (`-flto` on g++) particularly on older versions of the compiler. Since FAR++ uses fairly modern features of C++ and relies upon compilers implementing them efficiently always try the newest compiler that you have access to.

### Creating an array

Arrays in FAR++ are objects and are templated on the type of the array and the rank of the array. So creating an array is as simple as

```
#include <iostream>
#include "far.h"

int main(){
    far::Array<int,2> myArray;
}
```

This creates an integer rank 2 array, but it does not give it a size. The syntax looks very similar to that for `std::array`, but there the parameters are a type and a number of elements. Here it is a type and a rank for the array. Until you allocate the array it cannot be used. All FAR++ arrays are dynamically sized as the program runs. In Fortran terminology, they are “allocatable” arrays. You can give an array a size in various ways, following both C++ and Fortran approaches. The C++ style would be to pass a comma separated list of sizes to the constructor for the class

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> myArray(2,2);
}
```

This allocates the array to have 4 elements laid out as a 2x2 grid. You can now access elements by using the function call (round bracket) operator. Just like Fortran, the array elements are based from 1 by default. Unlike in C++ you do not have multiple brackets, but a single set of round brackets with the indices comma separated within them, again like Fortran. This operator returns a reference to the selected item of the array, so it can be used both to read and to write to the stored value

You can see the access working below

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> myArray(2,2);
    for (int j=1;j<=2;++j){
        for (int i=1;i<=2;++i){
            myArray(i,j)=i*j;
        }
    }
    std::cout << myArray << "\n";
}
```

FAR++ arrays can be printed directly to any ostream and will print in much the same way that Fortran prints arrays, element by element in the order of the underlying memory.

You can also allocate arrays using a Fortran like syntax with the “allocate” operator. This function takes the name of the array to allocate and then a list of sizes comma separated.

```
#include <iostream>
#include "far.h"
```

```
int main(){
  far::Array<int,2> myArray;
  allocate(myArray,2,2);
  for (int j=1;j<=2;++j){
    for (int i=1;i<=2;++i){
      myArray(i,j)=i*j;
    }
  }
  std::cout << myArray << "\n";
}
```

The equivalent code in Fortran would be

```
PROGRAM p
  IMPLICIT NONE
    INTEGER, DIMENSION(:,:), ALLOCATABLE :: myArray
    INTEGER :: i, j
    ALLOCATE(myArray(2,2))
    DO j = 1, 2
      DO i = 1, 2
          myArray(i,j) = i*j
        END DO
    END DO

    PRINT *, myArray
END PROGRAM p
```

You can see that the `allocate` syntax is different in detail, but similar in effect. You will also note that my nested loops are ordered so as to access the arrays in column major order - the first index to the array is updated by the inner loop and varies fastest. This is the correct way for accessing arrays in Fortran and is the correct default way of accessing FAR++ arrays. It is different to the normal way of accessing C++ arrays which is row major with the last index varying fastest. It is possible to configure FAR++ to use row major ordering, but normally you should access the elements of a FAR++ array in column major order, with the first index in the round brackets varying fastest.

Finally, you can allocate FAR++ arrays by construction and assignment in a C++ style

```
#include <iostream>
#include "far.h"

int main(){
    far::Array<int,2> A;
    A = far::Array<int,2>(3,4);
}
```

### Initialising an array

As you can see, you can give FAR++ arrays values element by element, but while this is the most important feature of them, it is perhaps the least interesting. As with Fortran, FAR++ gives you other options. The simplest is simply assigning a single value to the whole array. This is done exactly as it sounds, by using the assignment = operator.

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> myArray(2,2);
  myArray=14;
  std::cout << myArray << "\n";
}
```

FAR++ performs a type check to make sure that it is possible to assign the value on the right hand side of the = sign to the types stored in the array on the left hand side, but otherwise imposes no requirements. If you create a FAR++ array of a class that has `operator=` operators then you will be able to assign a value of that type to the array and it will be assigned to all elements.

FAR++ Arrays can be given a value by assigning another FAR++ array to them and, as in Fortran 2003 and newer, FAR++ arrays are automatically reallocated if another FAR++ array is assigned to them and they are either unallocated or allocated to be the wrong size. This means that FAR++ arrays are safe against buffer overruns when doing whole array assignment

The simplest functions for initialising an array are `zeros` and `ones`. These functions are not found in Fortran, but are designed by analogy with similar functions in other languages. These functions return an array, filled either with zeros or ones respectively. Because they return an array, they can be assigned to an array and it will be both allocated and initialised during the assignment

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,1> A;
  A=far::zeros(10); // Make a 10 element array of all zeros
  std::cout << A << "\n";

  A=far::ones(4); // Make a 4 element array of all ones
  std::cout << A << "\n";
}
```

The `zeros` and `ones` functions can be used to create arrays of any rank by comma separating the sizes, so to initialise rank 2 arrays you can use them as follows

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> A;
  A=far::zeros(3,3); // Make a 3x3 element array of all zeros
  std::cout << A << "\n";

  A=far::ones(5,5); // Make a 5x5 element array of all ones
  std::cout << A << "\n";
}
```

The reallocation behaviour is unchanged, but you can't change the rank of an array during assignment. If you have a rank 2 array then you have to provide 2 parameters to `zeros` or `ones`, similarly three parameters for a rank 3 array etc. etc.

There are some initialisation routines that always return a specific rank of array. The simplest example of this is `literal`. `literal` takes a comma seperated list of values and returns a rank 1 array containing copies of those values. The type of the returned array is determined by the first parameter in the list of parameters (although this can be overriden by passing a template parameter of the core type of the array to `literal`). Using `literal` to initialise a rank 1 array is exactly like using `ones` or `zeros`

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,1> myArray = far::literal(1,2,2,4);
  std::cout << myArray << "\n";
}
```

But since `literal` will always return a rank 1 array you have to use another function if you want to assign it to a higher rank array. The function is called `reshape` and works much like the Fortran function of the same name - you pass it an array of any given rank and a comma separated list of sizes and it will convert the source array values to the destination rank, by assuming that elements are laid out in memory in the same order in both the source and destination array.

```
#include <iostream>
#include "far.h"
```

```
int main(){
  far::Array<int,2> myArray = far::reshape(far::literal(1,2,2,4),2,2);
  std::cout << myArray << "\n";
}
```

The `reshape` function is one of the few places in FAR++ where you can get a runtime exception being generated by FAR++ itself. Since the sizes for the reshaped array can be specified at runtime FAR++ can only confirm that the source and the reshaped arrays are conformable at runtime. If they are not then a `std::out_of_range` error will be thrown. Also, please note that while this is not a concern when using it with functions like `literal`, `reshape` will make a copy of the source unless the source is a contiguous rvalue (just to clarify, the purpose of the reshape function is to make a copy of the source with a different rank and or shape, but if the array being reshaped is a temporary rvalue array like those from `literal` then as an optimisation the memory that holds the result is just moved to the destination)

The `literal` function really does create an rvalue `far::Array` object. It is an array literal in much the same way that `[1,2,2,4]` or `(/1,2,2,4/)` would be in Fortran, and it can be passed to functions etc. just as any other array can, so long as your function can handle rvalue parameters.

FAR++ implements move semantics for the `far::Array` classes, so assigning an rvalue array such as that generated by `literal` or `reshape` to an array will trigger an efficient move operation rather than causing a copy. Assigning one lvalue `far::Array` to another lvalue `far::Array` will cause a copy, and if needed a reassignment.

Another common function that works like `literal` is `linspace`. `linspace` is again an extension to Fortran. It takes three parameters: the start of a range of numbers, the end of the range of numbers and the number of elements in between, and it returns an array of the linearly spaced N values between the start and the end of the range. This is intended to partly replace the `implied do loop` feature of Fortran which is often used for this type of problem. It is important to note that the first two parameters to linspace(the start and end of the range) must be of the same type

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,1> myArray = far::linspace<int>(1,10,10);
  std::cout << myArray << "\n";
}
```

In this example, you can see that we explicitly gave a type template parameter to the `linspace` function. This specified what type the returned array should be, and almost all FAR++ functions that return an array without taking a source array take a kind parameter like this. By default `linspace`

returns an array of doubles. Because doubles are freely convertible to integers (with rounding) this code will work without this template parameter, but it will require that the results of the `linspace` function are copied into `myArray` rather than moved. Using the template parameter to make the generated array of the same type as the destination will prevent this copying. There is also an equivalent of `linspace` called `logspace` that spaces the points logarithmically between the lower and upper bounds

Finally, for ranks 1 to 3, it is possible to assign initializer lists to FAR++ arrays.

```
#include <iostream>
#include "far.h"

int main(){
  {
    far::Array<int,1> myArray = {1,2,3};
    std::cout << myArray << "\n";
  }
  {
    far::Array<int,2> myArray = {{1,2,3},{4,5,6}};
    std::cout << myArray << "\n";
  }
  {
    far::Array<int,3> myArray = {{{1,2,3},{4,5,6}},{{7,8,9},{10,11,12}}};
    std::cout << myArray << "\n";
  }
}
```

This could be extended to higher ranks but the syntax's usability rapidly reduces, so at present it is restricted to rank 3 or lower.

Sometimes you don't want to initialise an array with values from another source, but you want to take the size information from another array. In Fortran this is done by using the `mold` parameter to the `allocate` function, but in FAR++ it is done as a method on an array called `mold`.

```
#include <iostream>
#include "far.h"
#include <string>

int main(){

  far::Array<std::string,2> A(4,7);
  far::Array<int,2> B;
  B.mold(A);
```

```
  std::cout << far::shape(B) << "\n";

}
```

As you can see, A and B have completely incompatible types, but can still be used in a mold statement. The rank of the source and destination must match.

### Disposing of an array

FAR++ arrays do not need to be explicitly deallocated for memory safety. When a FAR++ array goes out of scope it automatically releases any memory that it is responsible for. Deallocating FAR++ arrays is only needed if you want to keep the array variable in scope but release the memory of the actual array. The main reason for doing this is to manually reallocate an array. As a safety feature Fortran gives a runtime error if you try to allocate an allocatable array that is already allocated and this behaviour is replicated in FAR++. Attempting to allocate an already allocate FAR++ array will raise `std::runtime_error`. We consider this to be a useful safety feature, preventing a user from accidentally changing the shape of an array in a way that might prevent correct operation of a code.

Deallocation is very simple and uses the `deallocate` function. Deallocate takes only allocated FAR++ arrays as parameters. Once they are deallocated their memory is released and the arrays can be allocated again.

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> myArray = far::reshape(far::literal(1,2,2,4),2,2);
  deallocate(myArray);
}
```

By analogy with Fortran, deallocating an unallocated array will cause a `std::runtime_error` to be raised and trying to explicitly allocate an array when it is already allocated will also cause a `std::runtime_error`. The restriction on allocation only applies to explicit allocation with the `allocate` command. Allocation caused by assigning a different sized FAR++ array will always allocate the destination to the correct size.

FAR++ adds an explicit `reallocate` command to deallocate an allocatable array and reallocate it to the new size. This function does NOT keep the content of the existing array, it simple avoids having to `deallocate` and then `allocate` the arrays explicitly. If the array is not initially allocated then an `std::runtime_error` will be raised when you call `reallocate`. This command is purely

designed to simplify the deallocate/reallocate sequence that is common in Fortran so has the same safety behaviour.

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> myArray;
  //reallocate(myArray,3,3); // This line will cause a runtime error
   ↪  because myArray is not allocated
  allocate(myArray,2,2); //This line will work because we are allocating an
↪  unallocated array
  //allocate(myArray,3,3); // This line will cause a runtime error because
   ↪  you cannot allocate an already allocated array
  reallocate(myArray,3,3); //This line will work because we are
↪  reallocating an allocated array
}
```

#### Whole array operations

Fortran and FAR++ both implement a wide range of whole array operations. We have already seen some of these, and they can only be used on arrays that have been allocated in one fashion or another.

**Assigning a single value to the whole array** You can assign every value in an array by simply assigning a value to is using =.

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> myArray(2,2); // Have to allocate array before
↪  assignment when assigning from non-array type

  myArray = 14; //Assign it
  std::cout << "Assign whole array to 14 gives : " << myArray << "\n";

}
```

Type checking is relaxed, so any type which can be assigned to the types stored in the FAR++ array can be assigned to the whole array even if it is not of the same type. This includes custom assignment operators

```
#include <iostream>
#include "far.h"

struct demo{
  int i;
  void operator=(int i){this->i=i;}//Assign operator
};

inline std::ostream& operator<<(std::ostream& os, const demo& obj) {
  os << obj.i << " ";
  return os;
}

int main(){
  far::Array<demo,2> myArray(2,2); // Have to allocate array before
↪  assignment when assigning from non-array type

  myArray = 14; //Assign it

  std::cout << myArray << "\n";

}
```

As you can see here we have created both an assignment operator and a output operator for my demo class. This means that we can assign an integer to the whole array by using the = operator and can print the whole array. This is generally the approach in FAR++ - all operators are delegated to the operators defined on the contained class of the array.

**Element generation functions** As well as implementing Fortran functions in FAR++ we have also implemented some C++ style interfaces where Fortran's interface misses some things that C++ programmers would expect. This includes the functions `generate`, `for_each` and `iota`. As will be discussed later, FAR++ does implement C++ style random access iterators, but performance can be lower than using FAR++'s internal approaches so for common actions like these we have implemented FAR++ style versions of the functions.

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> myArray(2,2);

    far::generate(myArray,[](){return 6;});
```

```
    std::cout << "Assigning by generation function that will always yield 6
    ↪  gives : " << myArray << "\n";

    far::for_each(myArray,[](int &i){i*=2;});
    std::cout << "Assigning using for_each and a function that multiplies
    ↪  every element by 2 gives : " << myArray << "\n";

    far::iota(myArray,1);
    std::cout << "Assigning using iota starting from 1 gives : " << myArray
    ↪  << "\n";

}
```

All three functions work much as their C++ versions do, but rather than passing an interator to the start and the end range you simply pass the whole array. `generate` then takes a function with no parameters, returning a type that can be assigned to an element of the array. The function is then called once for each element of the array. `for_each` takes an array and a function, this function taking a reference to the internal elements of the array. Once again the function is called on each element of the array. `generate` takes an array and a function and calls the function for each array element in turn, assigning the return value of the function to the element. `iota` takes an array and an initial value. The elements of the array are gone through in order, the array element is assigned the value that is passed and then that value is incremented using ++;

**Array operators** Another form of whole array operation is operators. All of the normal mathematical operators are implemented for arrays and they are applied element wise. When combining two arrays using any operator the effect is to loop over all elements of the arrays (for binary operators) or array (for unary operators) and produce a result for the combination of the two elements with the same position in the array.

This brings up an important point. FAR++, like Fortran, does not check that the operands to an operator have the same size, nor does it stop when the smaller operand's size is exceeded, so do not operate on arrays of unequal size. It is not sufficient that arrays have the same total number of elements, they must be the same size in each rank individually to guarantee correctness.

The available operators are

- A+B - Addition
- A-B - Substraction
- A*B - multiplication
- A/B - Division
- A%B - Modulo division

- -A - unary negation
- +A - unary positive
- A%B - modulo division
- {stream} » A - extraction operator
- {stream} « A - insertion operator
- A++ - increment operator
- ++A - increment operator
- A– - decrement operator
- –A - decrement operator
- A+= B - N increment operator
- A-= B - N decrement operator
- A*= B - N multiplication operator
- A/= B - N division operator
- A%=B - N modulo operator
- < - Less than operator
- > - Greater than operator
- <= - Less than equal to operator
- >= - Greater than equal to operator
- == - Equality operator
- != - Non equality operator
- || - Logical or
- | - bitwise or
- |= - bitwise or assignment
- && - logical and
- & - bitwise and
- &= - bitwise and assignment
- ^ bitwise xor
- ! - logical negation
- ~ - bitwise negation

All of the binary operators can take arrays for either of their operands, except for the extraction and insertion operators which can be applied to an array of streams, but creating such an array is tricky. So, for example, you can add two arrays together as follows

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,1> Array1 = far::linspace<int>(1,10,10);
  far::Array<int,1> Array2 = far::linspace<int>(10,1,10);
```

```
  std::cout << "Array 1 = " << Array1 << "\n";
  std::cout << "Array 2 = " << Array2 << "\n";
  std::cout << "Array 1 + Array 2 = " << Array1 + Array2 << "\n";
}
```

You can assign the result of a calculation like this to an array and it will automatically be allocated to be the correct size.

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,1> Array1 = far::linspace<int>(1,10,10);
  far::Array<int,1> Array2 = far::linspace<int>(10,1,10);
  far::Array<int,1> ArrayR; //Not allocated

  std::cout << "Array 1 = " << Array1 << "\n";
  std::cout << "Array 2 = " << Array2 << "\n";
  ArrayR = Array1 + Array2; //Allocated in assignment
  std::cout << "Array 1 + Array 2 = " << ArrayR << "\n";
}
```

It is also possible to have either operand be a simple variable and that simple variable will be applied to every value in the array

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,1> Array = far::linspace<int>(1,10,10);

  std::cout << "Array = " << Array << "\n";
  std::cout << "1 + Array = " << 1+Array << "\n";
}
```

There is no specific order for the operands. `1+Array` and `Array+1` are entirely equivalent. You can chain together as many operators as you like, although there are a few restrictions compared to Fortran that are worth mentioning

- Fortran compilers are allowed to make simplifications that C++ compilers aren't. In Fortran, the compiler knows that adding together the same array four times is the same as multiplying by 4. This is because in Fortran the compiler truly knows about arrays and knows what rules they follow since arrays are a core part of the languge. In FAR++, arrays are implemented as classes

with custom operators for things like addition and multiplication. Custom operators don't have to comply with the same rules as maths, so this kind of simplification won't be performed. So, when writing FAR++ code using whole array operations, you should manually simplify the maths as far as possible!

- Don't directly use `auto` to create a variable to hold the result of the calculation! Always assign the result to a specific array type. The other weakness of using C++ custom operators for whole array operations is that each operator is executed to completion before the previous operation is started. That means that if you implement the operators in the obvious way where `array2=array+array` works by taking two arrays and returning another array when you start chaining operators you are going to run into performance and memory usage problems. Say for example you have `result = (array1+array2)/array3;`. In a simple implementation that would calculate a whole array holding the result of `array1+array2` and then use that array and divide every element of that array by `array3`. For each operation a whole new array is created, so the memory usage will increase linearly with the number of operations in an expression, since these temporary arrays may not be removed until the evaluation of the expression stops. Similarly the performance will be poor because of the overhead of creating these temporaries etc. etc. So FAR++ doesn't do that. We instead create different classes that *represent* the operations and can then all be evaluated element by element and doesn't create temporaries. These classes are generically called `lazyArray` because they lazily execute the expressions that you build with them and they can be freely converted back into Arrays. However, while these `lazyArray` objects improve memory usage and performance substantially they have a problem - they have to capture the parameters that are passed to them somehow so that they can use them when wanted, and they do this by capturing references to their parameters. This causes a problem for any parameters in your expression that are rvalues because they cease to exist once the line that is being executed ends. That is fine if you capture the result of the calculation into an Array on the line that the expression is defined, but you can't keep a `lazyArray` and execute it later (this isn't quite true, but mostly you shouldn't). If you capture the result using `auto` then it will capture the `lazyArray` object rather than converting it into an `Array` and you will wind up with an unusable variable. If you have a situation where you really don't know the type of the returned object, then use the `toArray` function just before capturing the result with auto to convert the `lazyArray` to an `Array` of matched type. So, either follow the previous example of creating an array of the right type and rank to hold the result or use `toArray` and auto as shown below

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,1> Array1 = far::linspace<int>(1,10,10);
```

```
  far::Array<int,1> Array2 = far::linspace<int>(10,1,10);

  std::cout << "Array 1 = " << Array1 << "\n";
  std::cout << "Array 2 = " << Array2 << "\n";
  auto ArrayR = far::toArray(Array1 + Array2); //toArray converts the
  ↪  lazyArray to an Array so that ArrayR is a proper array
  std::cout << "Array 1 + Array 2 = " << ArrayR << "\n";
}
```

Mostly `lazyArray` objects return values rather than references because they represent calculated values from array expression. These objects can be used anywhere that FAR++ expects to read an array value, but not anywhere where FAR++ expects to write an array value. There are a few advanced places in FAR++ where `lazyArray` objects return references to data rather than values. Those types of `lazyArray` object can be used where FAR++ expects to write an array value.

**Passing arrays to and from functions** You can pass an array to a function exactly as you would expect.

```
#include <iostream>
#include "far.h"

void func(far::Array<double,2> &param1, far::Array<double,2> &param2){

  std::cout << param1 + param2 << "\n";

}

int main(){
  far::Array<double,2> Array1 = reshape(far::linspace<double>(0,1,9),3,3);
  far::Array<double,2> Array2 = reshape(far::linspace<double>(1,0,9),3,3);

  //Should print all 1s
  func(Array1,Array2);
}
```

Since FAR++ arrays will generally have substantial amounts of data in them it is *strongly recommended* that you pass them by reference rather than by value. If you want to pass array expressions (or, more technically lazyArray objects) then you can do that, so long as your parameter is a const reference. This is both equivalent to normal C++ behaviour with rvalues and with Fortran behaviour with array expressions only being allowed to be parameters to a function if the parameter is `INTENT(IN)`. You can also pass an array expression as a rvalue reference, but since it is not meaningful to modify an array expression, this has no benefits over passing it as a const reference.

```
#include <iostream>
#include "far.h"

void func(far::Array<double,2> const &param1, far::Array<double,2> const
↪  &param2){

  std::cout << param1 + param2 << "\n";

}

int main(){
  far::Array<double,2> Array1 = reshape(far::linspace<double>(0,1,9),3,3);
  far::Array<double,2> Array2 = reshape(far::linspace<double>(1,0,9),3,3);

  //Should count down from 2 to 1 in 0.125 increments
  func(Array1,Array2*2);
}
```

Here, this code only works becuse param2 is a const reference. We have made Param1 a const reference as well because we are not making changes to it. The given example call to func would work with that being a normal reference since param1 is not an array expression.

This approach works because the `lazyArray` objects can be freely converted to `Array` objects, so the `lazyArray` is converted to an `Array` before it is passed to the function. This does come with a memory overhead (because the entire array corresponding to the array expression is generated) and (if you are not using the whole of the `lazyArray`) some performance overhead, so it is also possible to pass the `lazyArray` without using this type conversion.

This isn't trivial for two reasons

- You will almost certainly want to be able to pass both arrays and array expressions rather than just expressions
- `lazyArray` objects aren't all of the same type. `lazyArrays`' types depend on what operations they perform, so a lazyArray to add two arrays is of a different type to a lazyArray to add three items, and both are different to one that adds two arrays and divides by a constant value. You want *any* of them to be passed to your array

The simplest solution is just to use C++ templating to allow any parameter to be passed to a function, but this gives you no security at all. You could accidentally pass a completely unsuitable type, an array of the wrong type, an array of the wrong rank or similarly unsuitable variables. To avoid this FAR++ includes *concepts* that match to arrays. The most common concept is the `arrayParameter` concept

```
#include <iostream>
#include "far.h"

template<far::arrayParameter<double,2> P1, far::arrayParameter<double,2>
↪  P2>
void func(P1 &param1, const P2 &param2){

  std::cout << param1 + param2 << "\n";

}

int main(){
  far::Array<double,2> Array1 = reshape(far::linspace<double>(0,1,9),3,3);
  far::Array<double,2> Array2 = reshape(far::linspace<double>(1,0,9),3,3);

  //Should count down from 2 to 1 in 0.125 increments
  func(Array1,Array2*2);
}
```

Concepts are a C++ 20 feature that allows you to create a template parameter that has to match certain conditions. In this case is has to be either an array or an array expression/lazyArray that returns values of type `double` and has rank 2. There are other concepts provided for arrays or lazyArrays of any rank for a specific type, of any type for a specific rank or for any rank or type. Of course, if you want to be able to pass other types for a parameter then simply use `typename` as in normal C++.

Using any of these options means that if a function is called with an array expression then the lazyArray object is passed rather than a new array being created, which avoids any memory overhead and may lead to a performance improvement, especially if you only use a small part of an array parameter.

Returning arrays from functions is simple, just return whatever you have and specify the return type as an array. `auto` return types are OK, so long as you are not returning an array expression. Once again use `toArray` if you want to return an array expression from a function with an auto return type.

**Array instrinsic functions** As well as operators on arrays, Fortran defines a set of intrinsic array functions. These are built in functions that take arrays as parameters and operated on them in some ways. They are split into three groups

- Elemental functions - These are functions that can take either arrays or simple values as parameters. When they are called with single values then they take those parameters and return a value that is calculated from those parameters. When they are called with arrays then the result is another array having the same rank and size as the array parameter with each element of

the returned array being the result of applying the same function to each element of the array parameter

- Transformational functions - These are functions that take array parameters and transform them in some way. Often, but not always this is by reduction in one or more directions. These functions can only act on arrays and may return either arrays or values
- Inquiry functions - These are functions that return properties about an array. They generally return either values or rank 1 arrays

There is a good list of the Fortran intrinsic functions at the Fortran Wiki. Not all of these functions has been replicated in FAR++, but all of the functions that are relevant to arrays have been.

All of the elemental functions, whether mathematical or not are implemented so that they return lazyArray objects. Transformational and inquiry functions that return arrays all return actual arrays.

Some Fortran functions take an optional `INTEGER` parameter describing the *kind* of the returned value. C++ handles types very different, and Fortran functions taking a kind parameter have an optional template parameter in FAR++ which should be the typename of the requested return type. If no template parameter is specified for these functions then the default kinds are *double* for floating point types, *int64_t* for integer types and *char* for character types. These functions are flagged with <kind> in their description. To change the kind of the return, simply put the typename for the return that you want in <> after the name of the function

As mentioned, there are some functions that work differently here to how they work in Fortran. This is mainly because Fortran has the option to do keyword calling of functions (i.e. calling with parameters specified by name rather than by position), and by combining this with optional parameters Fortran can have much richer approach to function signatures than C++ can have. Most Fortran functions don't really make use of this, and can be replicated by function overloads in C++, but in some places it is not possible to replicate the Fortran approach. Where this happens some functions have variant names to allow you to select between overloads that Fortran makes elegant through keyword calling.

To give an example, in Fortran, the `random_seed` function can be used to either get or set the random seed by specifying either a `get` or `put` parameter

```
CALL random_seed(get=value)
CALL random_seed(put=value)
```

Getting and setting random seeds use the same type, so cannot be dealt with as an overload based on type, and it is almost meaningless to want to both get and put the seed value, so you can't have a function with both parameters, so it is impossible to directly replicate this interface in C++.

In particular, many functions in Fortran have an *optional* parameter called `mask` that is used to specify which elements of array parameters should be included in the action of the function. Because a fair

number of these have interfaces that can't be used elegantly in C++ and to keep the interface consistent, all of the masked versions of functions where optional mask values are present have been renamed to have `_with\mask` added to their name. So, for example, if you want to sum up only those elements of an array that are greater than zero you would run

```
auto result = sum_with_mask(a,a>0);
```

This is the natural place to put the list of functions, but there are some features that we haven't discussed yet that these functions are connected with, notably the idea of *pointers*. We will discuss them later.

There is a very good (but not quite complete) list of the intrinsic procedures in Fortran here, but we will cover them briefly here.

- abs(A) - Take the absolute value of a real, integer or complex number
- achar<kind>(A) - Get the ascii value of a character as an integer value. Guaranteed to be ASCII even on systems that do not use ASCII (pointless nowadays, but Fortran guarantees this)
- acos(A) - Inverse cosine of a real or complex number
- acosh(A) - Inverse hyperbolic cosine of a real or complex number
- acospi(A) - Inverse hyperbolic cosine of real of complex numbers on [-1,1]
- aimag(A) - Returns the imaginary part of a complex number
- aint(A) - Converts a floating point number to an integer by truncation
- all(A) - Transformational function. Returns true if all elements of A are true
- all(A,dim) - Transformational function. Returns an array of parameters of 1 rank smaller than A, with the direction *dim* removed. For each remaining element, the value is true if all elements along *dim* are true and false otherwise. *dim* runs from 1 to rank
- allocated(A) - Returns true if an array variable is allocated and false if it is not
- anint(A) - Converts a floating point number to an integer by rounding to nearest integer
- any(A) - Transformational function. Returns true if any elements of A are true
- any(A,dim) - Transformational function. Returns an array of parameters of 1 rank smaller than A, with the direction *dim* removed. For each remaining el ement, the value is true if any elements along *dim* are true and false otherwise. *dim* runs from 1 to rank
- asin(A) - Inverse sine of a real or complex number
- asind(A) - Inverse sine of a real of complex number in degrees
- asinh(A) - Inverse hyperbolic sine of a real or complex number

- asinpi(A) - Inverse sine of a real or complex number with the result on [-1,1]
- associated(A) - Determines whether an array is pointing to another array
- associated(A,Target) - Determines whether an array is pointing to the specific array target
- atan(A) - Inverse tangent of a real or complex number
- atan(y,x) - Inverse tangent with two parameters
- atan2(y,x) - Inverse tangent with two parameters
- atan2d(y,x) - Inverse tangent with two parameters with the result in degrees
- atan2pi(y,x) - Inverse tangent with two parameters with the result on [-1,1]
- atand(a) - Inverse tangent with the result in degrees
- atand(y,x) - Inverse tangent with two parameters with the result in degrees
- atanh(A) - Inverse hyperbolic tangent of a real of complex number
- atanpi(A) - Inverse tangent with the result on [-1,1]
- atanpi(y,x) - Inverse tangent with the result on [-1,1]
- bessel_j0(A) - Bessel function of the first kind of order 0
- bessel_j1(A) - Bessel function of the first kind of order 1
- bessel_jn(n,A) - Bessel function of the first kind of order n. n must be an integer
- bessel_jn(n1,n2,x) - Returns a rank 1 array of all Bessel functions of the first kind of orders between n1 and n2 for the parameter x. X here must be a scalar
- bessel_y0(A) - Bessel function of the second kind of order 0
- bessel_y1(A) - Bessel function of the second kind of order 1
- bessel_yn(n,A) - Bessel function of the second kind of order n. n must be an integer
- bessel_yn(n1,n1,x) - Returns a rank 1 array of all Bessel functions of the first kind for orders between n1 and n2 inclusive for the parameter x. x must be a real scalar.
- bge(A,B) - Returns a boolean for whether A is greater than or equal to B in a bitwise sense. A and B must be integers
- bgt(A,B) - Returns a boolean for whether A is greater than B in a bitwise sense. A and B must be integers
- bit_size(A) - Returns the size of an object in bits. If A is an array then it returns the size of one element of the array

- ble(A,B) - Returns a boolean for whether A is less than or equal to B in a bitwise sense. A and B must be integers
- blt(A,B) - Returns a boolean for whether A is less than B in a bitwise sense. A and B must be integers
- btest(A,B) - Returns a boolean for whether the bit numbered B is set in A. A must be an integer. B=0 tests the least significant bit
- ceiling(A) - Returns the smallest integer greater than or equal to A
- f_char(A) - Returns the character corresponding to given integer value in the system's native text collating sequence. Optionally takes a template parameter for the type of the returned character. Called *char* in Fortran, but had to be renamed since it collides with a keyword in C++
- cmplx<kind>(A) - Returns a complex number, optionally taking a template parameter for the type of complex number to return. If A is real or integer then it sets the real part of the complex number. If A is complex then it sets both real and imaginary parts.
- cmplx<kind>(A,B) - Returns a complex number, optionally taking a template parameter for the type of the complex number to return. A and B must be real or integer and set the real and complex parts of the returned complex number
- conjg(A) - Returns the complex conjugate of the supplied complex number
- cos(A) - Returns the cosine of a real or complex array
- cosd(A) - Returns the cosine of a real or complex array with the argument in degrees
- cosh(A) - Returns the hyperbolic cosine of a real or complex array
- cospi(A) - Returns the cosine of a real or complex array with the argument on [-1,1]
- count(A) - Returns the number of elements of A that are true. A must be boolean
- count(A,dim) - Transformational function. Returns an array of parameters of 1 rank smaller than A, with the direction *dim* removed. For each remaining element, the value is the number of elements that are true along dimension dim
- cshift(A,shift) - Circularly shifts the array by shift places. If A is of rank > 1 then this will shift the array in dimension 1. Shift can either be an integer scalar or a rank 1 array. If it is a rank 1 array then it must have as many elements as A has in dimension 1.
- cshift(A,shift,dim) - Circularly shifts the array by shift places in direction dim. Shift can either be an integer scalar or a rank 1 array. If it is a rank 1 array then it must have as many elements as A has in dimension dim
- dble(A) - Convert an integer, real or complex to double precision. If complex, only the real part is converted

- dereference(A) - Takes an array of pointers and returns a lazyArray that returns references to the item that the pointers points to
- digits(A) - Return the number of significant sigits in the type of A
- dim(x,y) - Returns the maximum of x-y and zero
- dot_product(vector_a, vector_b) - The dot product of two vectors (must be rank 1). If the vectors are reals then this is the normal dot product, if the vectors are complex then this is the complex conjugate of vector_a multiplied with vector_b and summed, and if the vectors are bools then the result is any (vector_a && vector_b)
- dprod(x,y) - Returns the elementwise product of the elements of x and y with the result being double precision numbers
- dshiftl(i,j,shift) - conbined leftward bit shift of i and j by shift bits
- dshiftr(i,j,shift) - combined righward bit shift of i and j by shift bits
- eoshift(array, shift) - Shifts array in the first index by shift places. New values rotated into place are value initialised.
- eoshift(array, shift, dim) - Shifts array in the dim index by shift places. New values rotated into place are value initialised
- eoshift_with_boundary(array, shift, boundary) - Shifts array in the first index by shift places. If boundary is a constant then all new values are boundary. If boundary is an array then it should be one rank smaller than array and be of the size of array with the first dimension removed. This function is a change to the Fortran interface to add boundary calls
- eoshift_with_boundary(array,shift,boundary,dim) - Shifts array in the dim ndex by shift places. If boundary is a constant then all new values are boundary. If boundary is an array then it should be one rank smaller than array and be of the size of array with the dim dimension removed. This function is a change to the Fortran interface to add boundary calls
- epsilon(x) - Returns a nearly negligible number relative to 1
- erf(x) - Returns the error function of x
- erfc(x) - Returns the complementary error function of x
- erfc_scaled(x) - Returns the scaled complementary error function of x
- exp(x) - Exponential function of x (raises Euler number to x)
- exponent(x) - Return the exponent of a floating point number
- findloc<kind>(array,value) - Return an array containing the array position of the first element of array that compares true to value

- findloc<kind>(array,value,back) - Return an array containing the array position of the first element of the array that compares true to value if back is false, and the last element of the array that compares true to value if back is true
- findloc<kind>(array,value,dim) - Return an array of rank 1 lower than “array” each element containing the index in dimension dim of the first element in “array” that compares true to value
- findloc<kind>(array,value,dim, back) - Return an array of rank 1 lower than “array” each element containing the index in dimension dim of the first element in “array” that compares true to value if back is false and the index in dimension dim of the last element in “array” that compares true to value
- findloc_with_mask<kind>(array,value,mask) - Return an array containing the array position of the first element of array that both compares true to value and mask is true
- findloc_with_mask<kind>(array,value,mask,back) - Return an array containing the array position of the first element of array that both compares true to value and mask is true and back is false. If back is true then return an array containing the array position of the last element of array that both compares true to value and mask is true
- findloc_with_mask<kind>(array,value,mask,dim) - Return an array of rank 1 lower than “array” each element containing the index in dimension dim of the first element in “array” that compares true to value where mask is also true
- findloc_with_mask<kind>(array,value,mask,dim,back) - Return an array of rank 1 lower than “array” each element containing the index in dimension dim of the first element in “array” that compares true to value and mask is true if back is false and the index in dimension dim of the last element in “array” that compares true to value and mask is true
- f_float(i) - Convert an integer to a float
- floor<kind>(a) - Return the integer value of a rounded by flooring
- for_each(array,callable) - Not a Fortran function. Takes an array and a callable (function, lambda or class implementing operator()) that takes a single parameter that is a reference to a single element of array. The function is called once for each element of array, accessing each element in the native order of the array elements
- fraction(x) - returns that fractional part of x
- gamma(x) - returns the gamma function of x
- generate(array,callable) - Not a Fortran function. Takes an array and a callable (function, lambda or class implementing operator()). The function takes no parameters but returns a value that can be assigned to a single element of array. The callable is executed once for every element of

array, going through in the native order of the array elements and the result of the callable is assigned to the element of the array

- huge(x) - Returns the largest representable number of the same type as x
- hypot(x,y) - Returns the Euclidian distance of x and y (sqrt($x^{2+y}2$))
- iall(source) - Array reduction using bitwise and
- iall(source,mask) - Array reduction using bitwise and, only considering those values where mask is true
- iall(source,dim) - Array reduction using bitwise and along direction dim
- iall(source,dim,mask) - Array reduction using bitwise and along direction dim, only considering those values where mask is true
- iany(source) - Array reduction using bitwise or
- iany(source,mask) - Array reduction using bitwise or, only considering those values where mask is true
- iany(source,dim) - Array reduction using bitwise or along direction dim
- iany(source,dim,mask) - Array reduction using bitwise or along direction dim, only considering those values where mask is true
- ibclr(i,pos) - Set the pos-th bit of i to 0
- ibits(i,pos,len) - Return the “len” bits from i starting from pos and right shifted. All other bits are zero
- ibset(i,pos) - Set the pos-th bit of i to 1
- ieor(i,j) - Bitwise xor
- f_int<kind>(a) - Convert a to integer using truncation
- ior(i,j) - Bitwise or
- iota(array,value) - Not a Fortran function. Equivalent of C++ iota function. Fills the elements of array with values starting with value. After each assignment value is incremented with the ++ operator
- iparity(array) - Combine all of the elements of array using exclusive or
- iparity(array,mask) - Combine all of the elements of array where mask is true using exclusive or
- iparity(array,dim) - Combine the elements of array along dimension dim using exclusive or
- iparity(array,mask,dim) - Combine the elements of array where mask is true along dimension dim using exclusive or

- ishft(i,shift) - Shift the bits in i by shift places. If shift is negative then bits are shifted to the right, otherwise shift to the left
- ishiftc(i,shift) - Circularly shift the bits in i by shift places. Shift must be positive
- ishiftc(i,shift,size) - Circularly shift the rightmost shze bits of i by shift places. Shift must be positive
- is_contiguous(a) - Returns true if an array is contiguous (not a pointer to a slice or an array expression)
- kind<array> - Very different to the Fortran equivalent. This is a template expression that returns the type that an array's elements are made of
- lbound<kind>(x) - Returns an array of the lower bounds of a FAR++ array
- lbound<kind>(x,dim) - Returns the lower bound of a FAR++ array in dimension dim. Returns a value, not an array
- leadz(i) - Returns the number of zero leading bits of i
- linspace<kind>(min,max,N) - Extension to Fortran. Creates a rank 1 array of N linearly spaced values between min and max (inclusive)
- log(x) - Returns the natural logarithm of x
- logspace<kind>(min,max,N) - Extension to Fortran. Creates a rank 1 array of N logarithmically spaced values between min and max (inclusive)
- log_gamma(x) - Returns the logarithm of the gamma function for x
- log10(x) - Returns the logarithm to the base 10 of x
- log2(x) - Not a Fortran function. Returns the lograithm to the base 2 of x
- logical<kind>(l) - Returns the boolean equivalent of l
- maskl<kind>(i) - Returns values having the leftmost i bits set to 1 and all other bits set to 0
- maskr<kind>(i) - Returns values having the rightmost i bits set to 1 and all other bits set to 0
- matmul(matrix_a, matrix_b) - Returns matrix/matrix, matrix/vector or vector/matrix multiplication of the two parameters
- max(a,b,c,…) - Returns that maximum value of the parameters. If any of the parameters are arrays then the result will be an array
- maxexponent(x) - Returns the maximum exponent of type represented by x

- maxloc<kind>(array) - Return an array containing the array position of the largest element of array. If there are multiple elements with the largest value, the location of the first value is returned
- maxloc<kind>(array,back) - Return an array containing the array position of the largest element of the array. If there are multiple elements with the largest value, the location returned will be the first value if back is false and the last value if back is true
- maxloc<kind>(array,dim) - Return an array of rank 1 lower than “array” each element containing the index in dimension dim of the largest element in that dimension
- maxloc<kind>(array,dim, back) - Return an array of rank 1 lower than “array” each element containing the index in dimension dim of the largest element in array. If there are multiple elements of array that have the largest value then return the first element if back is false and the last element if back is
- maxloc_with_mask<kind>(array,mask) - Return an array containing the array position of the largest element of array where mask is true
- maxloc_with_mask<kind>(array,mask,back) - Return an array containing the array position of the largest element of array where mask is true. If there are multiple locations with the largest value and true mask value then if back is false then the position of the first element is returned and if back is false then the position of the last element is rturned
- maxloc_with_mask<kind>(array,mask,dim) - Return an array of rank 1 lower than “array” each element containing the index in dimension dim of the largest element in that dimension
- maxloc_with_mask<kind>(array,mask,dim,back) - Return an array of rank 1 lower than “array” each element containing the index in dimension dim of the largest element in array. If there are multiple elements of array with the largest value then the first element is returned if back is false and the last element if back is true
- maxval(array) - Return the largest value in array
- maxval(array,dim) - Returns a rank-1 array with each element containing the largest element in the removed dimension
- maxval_with_mask(array,mask) - Returns the largest value in array where mask is true
- maxval_with_mask(array, dim, mask) - Returns a rank-1 array with each element containing the largest element in the removed dimension where mask is true
- merge(t_source, f_source, mask) - Returns the merger of the two sources. If mask is true, return t_source, if mask is false return f_source

- merge_bits(i,j,mask) - Returns the merger of the bits from two integers under a mask integer. Where a bit in mask is 1, the bit state of the output is the bit state from i, where a bit in mask is 0, the bit state of the output is the bit state from j
- min(a,b,c,…) - Returns that minimum value of the parameters. If any of the parameters are arrays then the result will be an array
- minexponent(x) - Minumum value for the exponent of the type of X
- minloc<kind>(array) - Return an array containing the array position of the smallest element of array. If there are multiple elements with the smallest value, the location of the first value is returned
- minloc<kind>(array,back) - Return an array containing the array position of the smallest element of the array. If there are multiple elements with smallest value, the location returned will be the first value if back is false and the last value if back is true
- minloc<kind>(array,dim) - Return an array of rank 1 lower than “array” each element containing the index in dimension dim of the smallest element in that dimension
- minloc<kind>(array,dim, back) - Return an array of rank 1 lower than “array” each element containing the index in dimension dim of the smallest element in array. If there are multiple elements of array that have the smallest value then return the first element if back is false and the last element if back is true
- minloc_with_mask<kind>(array,mask) - Return an array containing the array position of the smallest element of array where mask is true
- minloc_with_mask<kind>(array,mask,back) - Return an array containing the array position of the smallest element of array where mask is true. If there are multiple elements with the smallest value where mask is true then if back is false the first element will be returned and if back is true then the last element will be returned
- minloc_with_mask<kind>(array,mask,dim) - Return an array of rank 1 lower than “array” each element containing the index in dimension dim of the smallest element in that dimension
- minloc_with_mask<kind>(array,mask,dim,back) - Return an array of rank 1 lower than “array” each element containing the index in dimension dim of the smallest element in array. If there are multiple elements of array with the smallest value then the first element is returned if back is false and the last element if back is true
- minval(array) - Return the smallest value in array
- minval(array,dim) - Returns a rank-1 array with each element containing the smallest element in the removed dimension

- minval_with_mask(array,mask) - Returns the smallest value in array where mask is true
- minval_with_mask(array, dim, mask) - Returns a rank-1 array with each element containing the smallest element in the removed dimension where mask is true
- mod(a,p) - remainder of division of a by p
- modulo(a,p) - a modulo p
- move_alloc(src, dest) - Moves the data associated with array dest to the array src. Does not copy data unless absolutely necessary
- mvbits(from, frompos, len, to, topos) - Copies the value of len bits, starting from position frompos from the variable from to the variable to. In to the bits are placed starting from position topos
- nearest(x,s) - Returns the nearest number after x in the direction of the sign of s. If s is positive then it returns the next distinct number after x and if s is negative then it returns the previous distinct number before x
- nint(x) - Return the nearest integer to x
- norm2(array) - Calculate the L2 norm of array. That is sum the squares of the values
- norm2(array,dim) - Return an array with rank one less than array. Each element of the array is the sum of the square of all of the values from the removed direction
- f_not(i) - Returns the bitwise boolean inverse of i. Just `not` in Fortran, but changed because of a collision with a C++ keyword.
- ones(n1,n2,n3,...) - Not a Fortran function. Returns an array of size(n1,n2,n3,...) with every element 1
- out_of_range(array, mold) - Elemental function. Returns a logical array saying if each value of `array` can be succesfully represented in the same type as `mold`. If an element of `array` is `NaN` or `Inf` then always returns false for that value
- out_of_range(array, mold, round) - Elemental function. Returns a logical array saying if each value of `array` can be succesfully represented in the same type as `mold`. If `round` is true then it when converting floating point types to integer types they are rounded to the nearest integer. If `round` is false then it is rounded towards zero. If an element of `array` is `NaN` or `Inf` then it always returns false for that value
- out_of_range<mold_type>(array) - out_of_range in C++ style with the mold specified as a template parameter
- out_of_range<mold_type>(array,round) - out_of_range in C++ style with the mold specified as a template parameter

- pack(array) - Take all of the elements of array and pack them into a rank 1 array in native memory order
- pack(array, mask) - Take all of the elements of array where mask is true and pack them into a rank 1 array in native memory order
- pack(array, mask, vector) - Create a rank 1 array with the same size as vector. For so long as there are elements in array where mask is true, packs those elements into the output array, but also increment a source location in vector. Once you have run out of elements in array where mask is true pack the remaining elements of vector after the incremented source location into the output. The result is all of the elements in array where mask is true followed by the last elements of vector
- parity(mask) - Combine all of the elements of mask using logical XOR
- parity(mask,dim) - Returns an array with the dimension dim removed. The elements along the removed dimension are combined with XOR
- popcnt(i) - Count the number of bits set in the integer i
- poppar(i) - Partiy of the bits in i
- precision(x) - Returns the decimal precision of the type of x
- product(array) - Return the product of the elements of array
- product_with_mask(array,mask) - Returns the product the elements of array where the corresponding element of mask is true
- product(array,dim) - Returns an array with dimension dim removed. The elements along the removed dimension are combined by multiplication
- product_with_mask(array, dim, mask) - Returns an array with dimension dim removed. The elements along the removed dimension are combined by multiplication if mask is true
- pow(array,power) - Not Fortran function. Returns the values in array to the power of power
- radix(array) - Return the radix of the type of array
- random_distribution(harvest,distribution) - Not a Fortran function. Takes an instance of a C++ RNG distribution object and uses it to fill the elements of harvest
- random_distribution(handle, harvest,distribution) - Not a Fortran function. Takes an instance of a C++ RNG distribution object and uses it to fill the elements of harvest using the random number generator referred to by handle
- random_normal(harvest,mean,stdev) - Not a Fortran function. Populates harvest with normally distributed numbers with specified mean and standard deviation. Care must be taken using this

function because there is hidden state that cannot be retrieved. If you need to be able to restart a sequence exactly from saved data you have to use random_distribution

- random_normal(handle, harvest,mean,stdev) - Not a Fortran function. Populates harvest with normally distributed numbers with specified mean and standard deviation. Random numbers are drawn from the generator referred to by handle. Care must be taken using this function because there is hidden state that cannot be retrieved. If you need to be able to restart a sequence exactly from saved data you have to use random_distribution
- random_number(harvest) - Fills harvest with random numbers uniformly selected between 0 and 1, including zero and excluding 1
- random_number(handle,harvest) - Fills harvest with random numbers uniformly selected between 0 and 1, including zero and excluding 1. Uses the random number generator referred to by handle
- random_seed_size(size) - Returns the number of int32_t elements that are needed to correctly seed the RNG
- random_seed_size(handle,size) - Retuns the number of int32_t elements that are needed to correctly seed the RNG referred to by handle
- random_seed_put(seed) - Seeds the default RNG with the specified seed. The seed should be a rank 1 array of type uint32_t and the size from random_seed_size.
- random_seed_put(handle, seed) - Seeds the RNG referred to by handle with the specified seed. The seed should be a rank 1 array of type uint32_t and the size from random_seed_size
- random_seed_put(istream) - Seeds the default random number generator from an istream. C++ RNGs can only be saved and restarted by serialising and deserialising them from a stream, so this behaviour is being passed through here
- random_seed_put(handle, istream) - Seeds the random number generator referred to by handle from an istream. C++ RNGs can only be saved and restarted by serialising and deserialising them from a stream, so this behaviour is being passed through here
- random_seed_get(ostream) - Saves the default random number generator state to an ostream. C++ RNGs can only be saved and restarted by serialising and deserialising them from a stream, so this behaviour is being passed through here
- random_seed_get(ostream) - Saves the default random number generator state to an ostream. C++ RNGs can only be saved and restarted by serialising and deserialising them from a stream, so this behaviour is being passed through here
- range(x) - Returns the decimal exponent range of the type of x

- rank(a) - Returns the rank of a
- real<kind>(a) - Converts a value or array to a real version, by default double precision
- reallocate(a,s1,s2,s3,...) - Not a Fortran function. Reallocates array a to have a new size. Reallocation is only possible on an allocated array and is simply a deallocate followed by an allocate
- reduce(array,operation) - Use the function operation to reduce array to a single value. The operation should take two members of array and return the reduced value. Operation is called pairwise with the accumulated value of the reduction and then new value to be included in the reduction. If array is of zero size then an exception is thrown
- reduce_with_mask(array,operation,mask) - Same as reduce(array,operation) but only the elements where mask is true are included in the reduction. If the array is of zero size or no elements of mask are true then an exception is thrown
- reduce_with_identity(array,operation,identity) - Same as reduce(array,operation), but if array is of zero size, return the value of identity
- reduce_with_mask_and_identity(array,operation,mask,identity) - Same as reduce_with_mask(array,operation,m but if the array is of zero size or no elements are mask are true, return the value of identity
- reduce(array,operation,dim) - Reduce array by removing direction dim and combining the elements along dim using the function operation. If the array is of zero size then an exception will be thrown
- reduce_with_mask(array,operation,dim,mask) - Same as reduce(array,operation,dim) but only the elements where mask is true are included in the reduction. If the array is of zero size or no elements of mask are true then an exception is thrown
- reduce_with_identity(array,operation,dim,identity) - Same as reduce(array,operation,dim), but if array is of zero size, return the value of identity
- reduce_with_mask_and_identity(array,operation,dim, mask,identity) - Same as reduce_with_mask(array,operation,mask) but if the array is of zero size or no elements are mask are true, return the value of identity
- reshape(array,s1,s2,s3...) - Take an array of any rank and produce a new array containing the same elements in the same order but with a set of sizes given by the additional s parameters. For example `reshape(a,100)` will reshape a into a rank 1 array with 100 elements and `reshape(a,10,10)` would reshape the same data into a 10x10 array
- reshape(array,size) - Take an array of any rank and produce a new array containing the same elements in the same order but with a set of sizes given by thesize parameter. Size is of type `std::array<int64_t,rank>` because the size of it must be known at compile time, so it

cannot be a FAR++ array. The behaviour is the same as the version that takes multiple parameters. `reshape(a,std::array<int64_t,1>(100))` converts the elements in a into 100 element rank 1 array and `reshape(a,std::array<int64_t,2>(10,10))` produces a 10x10 rank 2 array. There is a FAR++ wrapper for `std::array<int64_t,rank>` called far::constLiteral. `far::constLiteral(1,2,3,4)` is equivalent to `std::array<int64_t,4>(1,2,3,`

- reshape_with_pad(array,size,pad) - Take an array of any rank and produce a new array containing the same elements in the same order but with a set of sizes given by the size parameter. The size parameter is an `std::array<int64_t,rank>`. If there are more elements in the size described by the size parameter than are in the source array then remaining elements are populated from pad, starting from the first element of pad once the elements of array are exhausted. As an extension to the Fortran version of this function, if pad is a scalar then it that scalar value will be used for all elements after the elements of array are exhausted
- reshape_with_order(array,size,order) - Take an array of any rank and produce a new array containing the same elements in the same order but with a set of sizes given by the size parameter. The size parameter is an `std::array<int64_t,rank>`. Order should be a FAR++ array containing the order in which to fill the output ranks. If order is `literal(2,3,1)` for a rank 3 destination array then the array will be populated with the elements for array by filling up rank 2, then rank 3 and finally rank 1.
- reshape_with_pad_and_order(array,size,order,pad) - as reshape_with_order but when the elements of array are exhausted the elements of pad are used to fill the reshaped array. As an extension to the Fortran interface pad can be a scalar value, in which case all remaining elements of the result are filled with the value of pad once the elements of array are exhausted
- rrspacing(x) - Returns the reciprocal of the relative spacing of the type of x
- scale(x,i) - Scales x the exponent of x by i
- set_exponent(x,i) - Sets the exponent of x to be i
- shape<kind>(source) - Returns an array containing the shape of source. It has the same number of elements as the rank of source, with each element being the number of elements in that dimension of source
- shifta(i,shift) - Right shift the elements of i by shift places with arithmetic replacement of the left most bits
- shiftl(i,shift) - Left shift the elements of i by shift with zero replacement of the rightmost bits
- shiftr(i,shift) - Right shift i by shift with zero replacement of the leftmost bits
- sign(a,b) - Returns the magnitude of a with the sign of b
- sin(x) - Returns the sine of x

- sind(x) - Returns the sine of x with x given in degrees
- sinh(x) - Returns the hyperbolic sine of x
- sin(pi) - Returns the sine of x with x given on [-1,1]
- size<kind>(array) - Returns an array containing the shape of source. It has the same number of elements as the rank of source, with each element being the number of elements in that dimension of source
- size<kind>(array,dim) - Returns a single value for the number of elements in array in dimension dim
- sngl(a) - Returns a converted to single precision (float) values
- spacing(x) - Returns the spacing of values for the spacing of values for the type of x at values near x
- spread(source,dim,copies) - Returns an array of rank 1 greater than source with the extra rank inserted before direction dim and having copies elements to it. The values in source are extended to the new dimesion unchanged
- sqrt(x) - Returns the square root of x
- sum(array) - Returns the sum of the elements of array
- sum_with_mask(array,mask) - Returns the sum of the elements of array where mask is true
- sum(array, dim) - Returns an array with rank 1 lower than array with direction dim removed. Along the removed direction all elements are combined by summation
- sum_with_mask(array,mask,dim) - Returns an array with rank 1 lower than array with direction dim removed. Along the removed direction all elements where mask is true are combined by summation
- tan(x) - Returns the tangent of x
- tand(x) - Returns the tangent of x, where x is given in degrees
- tanh(x) - Returns the hyperbolic tangent of x
- tanpi(x) - Returns the tangent of x, where x is given on [-1,1]
- tiny(x) - Represents the smallest positive number that can be stored in the type of x
- trailz(x) - Returns the number of trailing bits in x
- transpose(x) - X must be a rank 2 array. Transposes the axes of x
- ubound<kind>(array) - Returns an array of the upper bounds of array

- ubound<kind>(array,dim) - Returns the upper bound in direction dim of array. Returns just a single value
- unpack(vector,mask,field) - Unpacks the elements of vector into an array having the same type as vector and the same shape as mask. Where mask is true the elements of vector are unpacked in order. For all elements where mask is false, the corresponding element of field is put into the output
- where(condition,action,parameter1,parameter2,parameter3,...) - An attempt to replicate Fortran's `where` construct. The first parameter is an array of conditions, followed by a callable (of some kind) that is called when condition is true, followed by a list of arrays that you want to be able to act on when the condition is true. The callable should take a single element (either by value or by reference) of all of the arrays passed as parameters and will be called with the position matched element from each parameter where an element of condition is true.
- where_elsewhere(condition,where,elsewhere,parameter1,parameter2,parameter3,...) - An attempt to replicate Fortan's where/elsewhere construct. Behaves like `where` but takes two callables. Where condition is true `where` is called with the matched elements from the parameters and where condition is false, `elsewhere` is called with the same parameters
- zeros<kind>(s1,s2,s3,...) - Returns an array with the specified sizes where every element is zero

**Random number generation** Having described the array functions from Fortran, there is one set of them that has had to be changed substantially - random number generation. At the simplest end, it works very much as it does in Fortran

```
#include <iostream>
#include "far.h"

int main(){
    float f;
    far::random_number(f);
    std::cout << "Random number: " << f << std::endl;

    far::Array<double,2> d(3,3);
    far::random_number(d);
    std::cout << far::gridPrint(d) << std::endl;
}
```

You call the `random_number` function with either a value or an array as the *harvest* and that value or array is filled with random numbers. The default RNG is the `std::mt19937` RNG, although this can be switched out in various ways. By default the RNG is seeded automatically based on the C++

`std::random_device` when the RNG is first called so the random sequence will be different every time.

Seeding the RNG is similar to Fortran in concept, but different in detail. You first request the size of the seed that is needed and then allocate an array large enough to hold the seed. Then call a `put` method to set the seed.

```
#include <iostream>
#include "far.h"

int main(){

    int64_t seed_size;
    //Get the size of the random seed Fortran style
    far::random_seed_size(seed_size);
    //Get the size of the random seed C++ style
    seed_size = far::random_seed_size();
    far::Array<uint32_t,1> seed(seed_size);
    seed=14;
    far::random_seed_put(seed);

    //Get the random seed
    far::Array<double,1> r(10);
    far::random_number(r);
    std::cout << r << std::endl;

    //Reset the random seed and get a new random number
    far::random_seed_put(seed);
    far::random_number(r);
    std::cout << r << std::endl;

    //The random number should be the same as the first one
}
```

Unfortunately, C++ RNGs don't return random seeds in the same format that they take them. Instead, the only way of storing an RNG is to output it to a stream. `random_seed_get` takes an `ostream` object and there is also an overload of `random_seed_put` that takes an `istream` object to allow setting the RNG from the result of `random_seed_get`.

```
#include <iostream>
#include <sstream>
#include "far.h"

int main(){
```

```
    //Get the random seed
    far::Array<double,1> r(10);
    far::random_number(r);
    std::cout << "Initial 10 random numbers: " << r << std::endl;

    //Now save the RNG
    std::cout << "Saving RNG\n";
    std::stringstream ss;
    far::random_seed_get(ss);

    far::random_number(r);
    std::cout << "Next 10 random numbers: " << r << std::endl;

    //Now restore the RNG
    std::cout << "Restoring RNG\n";
    ss.seekg(0);
    far::random_seed_put(ss);
    far::random_number(r);
    std::cout << "The regenerated next 10 random numbers: " << r <<
    ↪  std::endl;

}
```

In this case, the exact random numbers generated will be different every time because we aren't seeding the RNG with any specific state, but you are able to save the state of the RNG so that you generate the same sequence twice

As an extension to Fortran, there is also the concept of an `RNGHandle`. This is an object that encapsulates a random number generator engine and it's current state and can be used to allow unrelated sequences of random numbers. You can use an `RNGHandle` object as the first parameter to all of the random number functions.

```
#include <iostream>
#include "far.h"

int main(){

    int64_t seed_size;
    far::RNGHandle rng1, rng2;
    //Get the size of the random seed Fortran style
    far::random_seed_size(seed_size);
    //Get the size of the random seed C++ style
```

```
    seed_size = far::random_seed_size();
    far::Array<uint32_t,1> seed(seed_size);
    seed=14;
    far::random_seed_put(rng1, seed);
    far::random_seed_put(rng2, seed);

    //Get the random seed
    far::Array<double,1> r(10);
    far::random_number(rng1, r);
    std::cout << r << std::endl;

    //Reset the random seed and get a new random number
    far::random_number(rng2, r);
    std::cout << r << std::endl;

    //The random number should be the same as the first one
}
```

This example shows that seeding and generating random numbers works exactly the same with RNGHandle objects as it does with the global RNG. You might wonder why there are versions of random_seed_size that take a handle, and that is because it is possible to use template parameters to an RNGHandle to determine what RNG engine should be used to generate the random values.

```
#include <iostream>
#include <random>
#include "far.h"

int main(){

    int64_t seed_size;
    far::RNGHandle<std::minstd_rand0> rng1;
    far::RNGHandle<std::ranlux48> rng2;
    //Get the size of the random seed C++ style
    seed_size = far::random_seed_size(rng1);
    far::Array<uint32_t,1> seed(seed_size);
    seed=14;
    far::random_seed_put(rng1, seed);

    //Get this size using the Fortran interface
    far::random_seed_size(rng2, seed_size);
    far::reallocate(seed, seed_size);
    seed=14;
    far::random_seed_put(rng2, seed);
```

```
    //Get the random seed
    far::Array<double,1> r(10);
    far::random_number(rng1, r);
    std::cout << r << std::endl;

    //Reset the random seed and get a new random number
    far::random_number(rng2, r);
    std::cout << r << std::endl;
}
```

This example is very similar to the previous example, but now I have two RNGHandle objects templated on different uniform random bit generators, specifically the original 1988 minimal standard linear congruential generator of Lewis, Goodman and Miller, and the second on the 48 bit RANLUX generator from 1994 by Lüscher and James. Even though I seed both RNGs with the same seed value of 14, they generate entirely different sequences

As further extensions to Fortran there is a function `random_normal`. This function takes a mean and standard deviation as well after the harvest parameter.

```
#include <iostream>
#include "far.h"

template<far::anyRankArray<float,double> param>
void mean_and_stdev(param& a){
    double mean;
    mean = far::sum(a) / (double)a.size();
    double stdev;
    stdev = far::sqrt(far::sum(far::pow(a - mean, 2.0)) /
↪   (double)a.size());

    std::cout << "Mean: " << mean << std::endl;
    std::cout << "Standard deviation: " << stdev << std::endl;
}

int main(){

    far::Array<float,2> a(3,3);

    // Fill the array with random numbers from a normal distribution
    // with mean 0 and standard deviation 1
    far::random_normal(a, 0.0f, 1.0f);
    std::cout << far::gridPrint(a) << std::endl;
```

```
    far::reallocate(a,1000,1000);
    far::random_normal(a, 0.0f, 4.0f);
    mean_and_stdev(a);

}
```

Another extension from Fortran there is also `random_distribution` function. This works much like `random_number` and `random_normal`, but takes a parameter after the `harvest` that is a C++ `RandomNumberDistribution` object or a FAR++ array of `RandomNumberDistribution` objects. The object is used to generate random numbers for each element of the harvest. As a simple example, consider sampling from a Weibull distribution.

```
#include <iostream>
#include "far.h"

int main(){

    far::Array<float,2> a(3,3);
    std::weibull_distribution<float> dist(1.0f, 1.0f);

    // Fill the array with random numbers from a weibull distribution
    // with shape 1 and scale 1
    far::random_distribution(a, dist);

    std::cout << far::gridPrint(a) << std::endl;

}
```

One final extension to Fortran's model is `random_array`. This function takes a set of sizes or `Range` objects (described later) describing the array to be returned, creates an array of a rank matching the number of parameters given, allocates it with the specified bounds and populates it with uniform random numbers. This function returns an actual array to make sure that you cannot accidentally create inconsistent results by evaluating a lazyArray multiple times.

```
#include <iostream>
#include "far.h"

int main(){
    far::Array<double,2> arr = far::random_array(5,7);
    std::cout << far::gridPrint(arr) << std::endl;
}
```

`random_normal`, `random_distribution` and `random_array` can also have an `RNGHandle`

object passed as the first parameter to sample numbers from a specific RNG.

**Custom array bounds** Another element of Fortran's arrays is that they can have a user specified lower bound, rather than always starting from one or always starting from zero. This is replicated in FAR++ and such bounds are specified as a Range object. Anywhere that you can use a number to specify the size of an Array you can use a Range object to specify custom upper and lower bound insted, so for example

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,1> A(far::Range(-1,8));
  A = far::linspace<int>(1,10,10);
  for (int i=far::lbound(A)(1);i<=far::ubound(A)(1);++i){
    std::cout << i << " : " << A(i) << "\n";
  }
}
```

In this case the array A is created with the custom bounds (-1,8) and the loop shows that you can access the elements using the custom bounds. You will notice that we have had to split defining the array and assigning it values onto two lines. This is a C++ restriction that is not present in Fortran but cannot be avoided. You can allocate an array by assigning an array to it on the line where it is defined (in which case it will always have the bounds of the source array, or 1 for arrays returned by functions like `linspace`) or you can assign it a size on the line when you define it (which can include custom lower bounds) and then you have to assign value on another line. *NOTE* because of the automatic reallocation behaviour when assigning to an array with custom lower bounds you must be *sure* that the array that you are assigning from has the same number of elements as the destination, otherwise the destination will be reallocated and will gain the bounds of the source array.

There is one difference to Fortran when you pass arrays to functions. In Fortran, sometimes arrays passed to functions retain their custom lower bounds, sometimes they default to a lower bound of 1 and the user can optionally specify a lower bound for arrays as they enter a function separately to the lower bounds attached to an array. This is often confusing in Fortran and would be hard to replicate in C++, so it is not replicated by FAR++. In FAR++ arrays passed to functions always retain their lower bounds.

```
#include <iostream>
#include "far.h"

void func(far::Array<double,1> &A){
```

```
  for (int i=far::lbound(A)(1);i<=far::ubound(A)(1);++i){
    std::cout << i << " : " << A(i) << "\n";
  }
}

int main(){
  far::Array<double,1> A(far::Range(-1,8));
  A = far::linspace(1,10,10);
  func(A);
}
```

Higher rank arrays are created in exactly the same way

```
#include <iostream>
#include <iomanip>
#include "far.h"

int main(){
  far::Array<int,2> A(far::Range(-1,1),far::Range(-1,1));
  A = far::reshape(far::linspace(1,9,9),3,3);
  for (int j=far::lbound(A,2);j<=far::ubound(A,2);++j){
    for (int i=far::lbound(A,1);i<=far::ubound(A,1);++i){
      std::cout << i << ", " << j << " : " << A(i,j) << "\n";
    }
  }
}
```

In the example pack there are also calls to setw and setfill in the print statement, these are just there to make the output easier to read.

You can freely mix specifying sizes with Range objects and with numbers

```
#include <iostream>
#include <iomanip>
#include "far.h"

int main(){
  far::Array<int,2> A(far::Range(-1,1),3);
  A = far::reshape(far::linspace(1,9,9),3,3);
  for (int j=far::lbound(A,2);j<=far::ubound(A,2);++j){
    for (int i=far::lbound(A,1);i<=far::ubound(A,1);++i){
      std::cout << i << ", " << j << " : " << A(i,j) << "\n";
    }
  }
}
```

**Array slices** The final major element of Fortran's arrays is *array slices*. The idea of an array slice is that you can represent a subsection of an array and use it and manipulate it as if it was an array. Array slices are different views onto an array and do not cause copies of data without you explicitly assigning to another array. Array slices are again done with Range objects.

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,1> A = far::linspace<int>(1,10,10);

  std::cout << "Whole array = " << A << "\n";
  std::cout << "Elements 2 to 4 = " << A(far::Range(2,4)) << "\n";
  ↪  //Fortran equivalent A(2:4)
  std::cout << "First 2 elements = " << A(far::Range(1,2)) << "\n"; //
  ↪  Fortran equivalent A(1:2)
  std::cout << "Start of array to element 2 = " << A(far::fromLB(2)) <<
  ↪  "\n"; // Fortran equivalent A(:2)
}
```

This shows creating a slice and printing it and their Fortran equivalents. You use a Range object with two parameters to specify a range between two specific points and there are also special versions for Fortran one sided ranges. These specials are

- far::fromLB(n) - From lower bound to specified value. Fortran (:n)
- far::toUB(n) - From specified value to upper bound. Fortran (n:)
- far::Range() - From lower bound to upper bound. Fortran(:)
- far::LBtoUB() - From lower bound to upper bound. Also Fortran (:)

The last two seem a bit pointless because it is equivalent to just using the array without a slice at all, and that is mostly true for rank 1 arrays, but the purpose for rank 2 arrays is clear - you might well want all of some ranks while restriction to subsections of others. Getting slices of higher rank arrays is the same

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> A = reshape(far::linspace<int>(1,25,25),5,5);

  //far::gridPrint is a method for printing a rank 2 array with some
  ↪  information about the bounds
  std::cout << far::gridPrint(A) << "\n\n";
```

```
  //Now print just a slice
  std::cout << far::gridPrint(A(far::Range(2,4),far::Range(2,4))) << "\n";

}
```

`gridPrint` is a built in feature of FAR++ that prints small rank 2 arrays with information about the indices of each element printed. Here, the output is

```
    |  1 |  2 |  3 |  4 |  5 |
    -----------------------
  1 |   1|   6|  11|  16|  21|
  2 |   2|   7|  12|  17|  22|
  3 |   3|   8|  13|  18|  23|
  4 |   4|   9|  14|  19|  24|
  5 |   5|  10|  15|  20|  25|

    |  1 |  2 |  3 |
    ----------------
  1 |   7|  12|  17|
  2 |   8|  13|  18|
  3 |   9|  14|  19|
```

This shows two things. First it shows that as expected the central section of the array is printed the second time (2 to 4 in each direction inclusively). Secondly, it shows that the array subection has indices from 1 to 3. i.e. it has lost the knowledge that it was 2 to 4 in the original array. This is important when you assign the array subsection to another array and trigger allocation. You can do that very easily

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> A = reshape(far::linspace<int>(1,25,25),5,5);
  far::Array<int,2> result;

  std::cout << far::gridPrint(A) << "\n\n";

  //This creates a new array holding the values in the selected device
  //This is the first point at which a copy of the data is made!
  result = A(far::Range(2,4),far::Range(2,4));
```

```
  std::cout << far::gridPrint(result) << "\n";

}
```

This code assigns an array subsection to another array, which finally copies the data into that other array. Before then, the array slice was just another view onto the same data in the original array. You can easily see that by using an array subsection on the left of an operation

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> A = reshape(far::linspace<int>(1,25,25),5,5);

  std::cout << far::gridPrint(A) << "\n\n";

  A(far::Range(2,4),far::Range(2,4)) = -5; //Set all values in the slice to
↪  -5

  std::cout << far::gridPrint(A) << "\n";

}
```

As you can see, you can assign values to a range, in this case assigning -5 to the section (2:4,2:4), and this is generally the case. Anywhere that you can use a FAR++ array you can use a FAR++ array slice.

While much less commonly used than the normal slices, Fortran (and hence FAR++) array sections can include a stride. That is an array subsection that is not adjacent elements, but elements that are separated by a fixed value in each dimension. This is done simply by adding a third value to the Range object for the distance between stride elements.

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> A = reshape(far::linspace<int>(1,25,25),5,5);

  std::cout << far::gridPrint(A) << "\n\n";

  A(far::Range(2,4,2),far::Range(2,4,3)) = -5;

  std::cout << far::gridPrint(A) << "\n";
```

```
}
```

This example again sets elements in the range (2:4,2:4) to be -5, but now with a 2 stride in dimension 1 and a 3 stride in dimension 2. This means that in dimension 1, 2 and 4 are changed (because 2+2=4, still within the specified range) and in dimension 2 only 2 is changed (because 2+3=5, outside the range). The output is thus

```
    |  1 |  2 |  3 |  4 |  5 |
    -----------------------
  1 |   1|   6|  11|  16|  21|
  2 |   2|   7|  12|  17|  22|
  3 |   3|   8|  13|  18|  23|
  4 |   4|   9|  14|  19|  24|
  5 |   5|  10|  15|  20|  25|

    |  1 |  2 |  3 |  4 |  5 |
    -----------------------
  1 |   1|   6|  11|  16|  21|
  2 |   2|  -5|  12|  17|  22|
  3 |   3|   8|  13|  18|  23|
  4 |   4|  -5|  14|  19|  24|
  5 |   5|  10|  15|  20|  25|
```

As you can see (2,2) and (4,2) are the only elements changed.

Strides can be negative, in which case the range must also be reversed, so it now goes `Range(upper_bound,lower_bound,stride)`.

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> A = reshape(far::linspace<int>(1,25,25),5,5);

  std::cout << far::gridPrint(A) << "\n\n";
  std::cout << far::gridPrint(A(far::Range(4,2,-1),far::Range(4,2,-1))) <<
  ↪  "\n";

}
```

This now prints the centre section of the array reversed in both directions

```
    |  1 |  2 |  3 |  4 |  5 |
    ----------------------
  1 |   1|   6|  11|  16|  21|
  2 |   2|   7|  12|  17|  22|
  3 |   3|   8|  13|  18|  23|
  4 |   4|   9|  14|  19|  24|
  5 |   5|  10|  15|  20|  25|

    |  1 |  2 |  3 |
    ---------------
  1 |  19|  14|   9|
  2 |  18|  13|   8|
  3 |  17|  12|   7|
```

Negative strides other than -1 can be used and work as do positive strides but with the direction reversed.

Sometimes you want to keep around a reference to a subsection of an array (or a whole array) and you can do that with the `pointTo` method of an array

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> A = reshape(far::linspace<int>(1,25,25),5,5);
  far::Array<int,2> dest;

  std::cout << far::gridPrint(A) << "\n\n";
  //This does NOT copy the slice. dest is now a pointer to the slice of A
  dest.pointTo(A(far::Range(2,4),far::Range(2,4)));

  dest = -5; //Set all values in the slice to -5

  //Print A, the (2:4,2:4) section will have been changed
  std::cout << far::gridPrint(A) << "\n";
}
```

pointTo is the equivalent of the Fortran => operator and causes a FAR++ array to become a reference to another array. You can point a FAR++ array to any other array (of the same type and rank) or any array slice (of the same type and rank). Arrays that are acting as pointers are no different to any other arrays from a type perspective, so can be used exactly like other Arrays. There is one important reason why

FAR++ pointer arrays are different to Fortran pointer arrays. Fortran pointers can be allocated to have lifetimes that are decoupled from their scopes by `ALLOCATEing` pointer arrays. In FAR++ this is not possible (directly) - pointers are simply different views onto normal allocatable arrays. An array that is currently a pointer can be deallocated and then allocated, but it then becomes a normal allocatable array and the memory associated with it is deallocated as soon as it goes out of scope.

There is one wrinkle when dealing with array slices, that is a fundamental part of the C++ standard. When you initalise a variable with a value returned from a function on the line where it is created then a process called Named Return Value Optimisation (NRVO) may take place. Under NRVO if you are assigning a value to a variable of the same type on the same line where the left hand side is being created the object that is used to initalise the variable is directly constructed in the memory that will hold the variable being assigned to rather than it being copied explicitly. None of the custom operators for copying or moving will trigger. Mostly this isn't a problem in FAR++ because one is either assigning an array to another array (in which case NRVO will happen and be correct), or one is assigning a lazyArray to an Array, in which case NRVO cannot happen because it is assignment between different types. Array slices and array pointers are the one exception to this. This is because when you slice an array it returns an array pointer to the slice.

Normally, when you assign an array pointer to an array the array is allocated and a copy of the array that the pointer points to is made, but when you assign an array pointer to an array on the line where the array is defined, you may instead wind up with an array pointer. We say "may" because NRVO is an optional optimisation that a compiler may or may not choose to make. To make this clearer

```
int main(){
  far::Array<int,2> A = far::reshape(far::linspace(1,25,25),5,5);
  far::Array<int,2> B = A(far::Range(2,4),far::Range(2,4));
}
```

In this code, B may be either a copy of the section of A, or a pointer to the section of A, depending on whether the compiler has chosen to perform NRVO. This is because taking a slice of an array technically returns a pointer array to the slice, which will be copied to the source array without NRVO, but simply constructed in place if it does occur. This is an example of one of only two optimisations that is permitted in C++ to optimise out side effects.

```
int main(){
  far::Array<int,2> A = far::reshape(far::linspace(1,25,25),5,5);
  far::Array<int,2> B;
    B = A(far::Range(2,4),far::Range(2,4));
}
```

In this code, B is guaranteed to be a copy of the relevant section of A

```
int main(){
```

```
  far::Array<int,2> A = far::reshape(far::linspace(1,25,25),5,5);
  far::Array<int,2> B;
    B.pointTo(A(far::Range(2,4),far::Range(2,4)));
}
```

In this code B is guranteed to be a pointer to the section of A.

Just as `toArray` helps with using `auto` to store `lazyArray` objects, FAR++ defines helper functions `forceArray` and `forcePointer` that can be used to avoid this ambiguity, but it is recommended that you don't assign a slice to a variable on the line that the destination is defined.

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> A = far::reshape(far::linspace(1,25,25),5,5);
  far::Array<int,2> B =
↪   far::forceArray(A(far::Range(2,4),far::Range(2,4)));
  B=14; //This will not affect A;
  std::cout << far::gridPrint(A) << "\n\n";

  far::Array<int,2> C =
↪   far::forcePointer(A(far::Range(2,4),far::Range(2,4)));
  C=14; //This will affect A
  std::cout << far::gridPrint(A) << "\n\n";
}
```

As an extension to Fortran, you can take a slice of the result of an array operation as well as taking a slice of a full array (although it is not possible to point a pointer to the result of an array operation)

```
#include <iostream>
#include "far.h"

int main(){
  far::Array<int,2> A = -reshape(far::linspace<int>(1,25,25),5,5);
  std::cout << far::gridPrint(A) << "\n\n";

  std::cout <<
   ↪   far::gridPrint(far::abs(A)(far::Range(2,4,-2),far::Range(2,4,-2))) <<
   ↪   "\n";
}
```

This example creates an array of negative numbers, runs it through `far::abs` to produce the positive versions of the numbers and then slices the result of that to get a 2x2 array of the centre section,

reversed with a stride of 2. The value of this is debatable, but it was easy enough to implement, so it is included as an option in FAR++.

One final point to note that you don't have to put a `Range` object into every position in the round brackets when slicing an array. You can just put a numerical index into any of the positions and that will slice the array at just that index. There is an important difference between specifying just a value and `Range(value,value)` which is that if you put in a single value for an array specification then you actually *reduce the rank of the array*. So for example

```
A(Range(1,1),Range());
```

gives you a 1xN array as the result of the slice

```
A(1,Range())
```

gives you a rank 1 array of size N. You can reduce as many ranks as wanted in a single slice like this, with all directions subscripted with a simple value removed from the resulting array. Although the syntax is not particularly readable, as an extension to Fortran you can slice a slice by using a second set of brackets after the first slice

```
#include <iostream>
#include "far.h"

int main(){
    far::Array<int,3> A = reshape(far::linspace<int>(1,27,27),3,3,3);

    std::cout << "\nSize of slice (Range(),Range(1,1),Range()) is " <<
    ↪  far::shape(A(far::Range(),far::Range(1,1),far::Range())) << "\n\n";
    std::cout << "Size of slice (Range(),1,Range()) is " <<
    ↪  far::shape(A(far::Range(),1,far::Range())) << "\n\n";

    far::Array<int,2> Slice = A(far::Range(),2,far::Range());
    std::cout << "Slice is \n" << far::gridPrint(Slice) << "\n\n";

    far::Array<int,1> Slice1D;
    Slice1D.pointTo(Slice(2,far::Range(1,2)));
    std::cout << "Slice(2,1:2) is " << Slice1D << "\n\n";

        Slice1D.pointTo(Slice(far::Range(3,1,-1),2));
        std::cout << "Slice(3:1:-1,2) " << Slice1D << "\n\n";

    std::cout << "Slicing straight from original array " <<
    ↪  A(far::Range(3,1,-1),2,2) << "\n\n";
```

```
    std::cout << "Slice of slice notation " <<
    ↪  A(far::Range(),2,far::Range())(far::Range(3,1,-1),2) << "\n\n";
}
```

The final thing that can be done with a pointer is *pointer remapping*. In this, an array or contiguous section of an array is remapped to have a different rank and or bounds. This is done with extra index specification parameters to the `pointTo` function. The index specifications must be suitable for the rank of the array that is being pointed at the target and may be either single values or `Range` objects, but the target can now be any array or continuous array slice. If you try and remap point to a non contiguous array then `std::runtime_error` will be raised.

```
#include <iostream>
#include "far.h"

int main(){
    far::Array<int,3> A = reshape(far::linspace<int>(1,60,60),3,4,5);

    std::cout << "A is a 3x4x5 array of the elements from 1 to 60\n";

    far::Array<int,2> B;
    B.pointTo(A,12,5);
    std::cout << "Remapping A to 12x5 array\n";
    std::cout << far::gridPrint(B) << "\n\n";

    B.pointTo(A,4,15);
    std::cout << "Remapping A to 4x15 array\n";
    std::cout << far::gridPrint(B) << "\n\n";

    B.pointTo(A,far::Range(-6,5),5);
    std::cout << "Remapping A to 12x5 array with custom bounds\n";
    std::cout << far::gridPrint(B) << "\n\n";

    std::cout << "Remapping A(1:3,1:4,2) to a 3*4 array\n";
    std::cout << "This works because it is a contiguous array\n";
    B.pointTo(A(far::Range(1,3),far::Range(1,4),2),3,4);
    std::cout << far::gridPrint(B) << "\n\n";

    std::cout << "Remappying A(1:3,1:4,5) to a 12 element rank 1 array\n";
    far::Array<int,1> C;
    C.pointTo(A(far::Range(1,3),far::Range(1,4),5),12);
    std::cout << C << "\n\n";

    far::Array<int,2> D;
```

```
    D.pointTo(C,4,3);
    std::cout << "Remapping 1D slice back to 4x3 array\n";
    std::cout << far::gridPrint(D) << "\n\n";

    std::cout << "Using remapped pointer to set (1:3:2,1:2)*=-1 in reformed
    ↪  4*3 slice\n";
    D(far::Range(1,3,2),far::Range(1,2))*=-1;
    std::cout << "Reformed slice after modification\n";
    std::cout << far::gridPrint(D) << "\n\n";
    std::cout << "Original array after modification\n";
    std::cout << far::gridPrint(A(far::Range(),far::Range(),5)) << "\n";
}
```

The previous example shows what can be done with pointer remapping. It remaps a rank 3 array to a rank 2 array, with a variety of sizes and bounds. We then remap a contiguous slice of the original array to a rank 2 array and then remap the rank 2 array down to a rank 1 array. The rank 1 array is then remapped back up to a different shaped rank 2 array. Once I have that rank 2 array from all the sequence of remapping I multiply a slice of the elements by -1, showing that this change propagates correctly back to the original data in the original array.

#### C++ features

While FAR++ is mainly intended to be used as a self contained “Fortran-Like” library, it does have to interact with C++ code so it does have features that are based on C++. We have already seen simple functions like `for_each`, `generate` and `iota` that are more based in C++ than Fortran, but FAR++ arrays and lazyArrays are also C++ STL containers, implementing forward and backward random access iterators. The iterators for FAR++ arrays simply go through the items in the array in the order in which they are stored. You can use the iterators for any purpose where normal iterators can be used

```
#include <iostream>
#include "far.h"

int main(){

    Array<int,2> a = far::reshape(far::linspace(1,5*6,5*6),5,6);
    for (auto &el:a){
      el*=2;
    }
    std::cout << far::gridPrint(a) << "\n";

}
```

This code goes through all of the elements of the array and multiplies each element by 2. It is entirely equivalent in effect to simply typing `a*=2`, although performance can be lower when using iterators (measured at up to about 20% slower). Because of the potential for lower performance, generally there is no reason to use the iterators when there are FAR++ methods to achieve things. On the other hand, the fact that all FAR++ arrays are iterable allows you to do things like this

```
#include <iostream>
#include "far.h"

int main(){

    far::Array<int,2> a = far::reshape(far::linspace(1,5*6,5*6),5,6);
    for (int i=1;i<=5;i++){
        std::sort(a(i,far::Range()).begin(),a(i,far::Range()).end(),[](int
        ↪  a, int b){return a>b;});
    }
    std::cout << far::gridPrint(a) << "\n";

}
```

This code sorts each row of the array independently of the other rows using the ability to get iterators over array slices. Mostly, it is expected that speed critical parts of a code using FAR++ will use FAR++'s "native" interface rather than the C++ STL interface. If you really need to have a very high performance STL-like interface to FAR++ arrays then you can create `contiguousArray` types. Contiguous arrays disable many of the mathematical operations used to allow slices and strides in FAR++, so they do not slow down when accessed through iterators.

```
#include <iostream>
#include "far.h"

int main(){

    far::contiguousArray<int,2> a =
↪   far::reshape(far::linspace(1,5*6,5*6),5,6);
    for (int i=1;i<=5;i++){
        std::sort(a(i,far::Range()).begin(),a(i,far::Range()).end(),[](int
        ↪  a, int b){return a>b;});
    }
    std::cout << far::gridPrint(a) << "\n";

}
```

Contiguous arrays are created in the same way as normal arrays but with a different type name. They

can be used anywhere Arrays are, but if you pass them to a function that explicitly requires an `Array` type rather than using the `arrayParameter` concept it will cause a copy to occur. You should not use contiguous arrays unless you want to use iterators, since they have no other real advantages over normal arrays. If you want temporary access to the data in a FAR++ array as a C++ bare pointer that is covered later.

FAR++ `lazyArrays` (representing array expressions) also implement const forward and backward iterators, so you can use them anywhere you could use an STL container as well. There is one problem - by default lazyArrays don't work properly except on the line where you define them. This is normally not a problem since mostly lazyArrays are used as rvalues to pass to functions or stored to array variables, but when you want to use their iterators it is quite annoying because you need to be able to call `.begin` and `.end` methods on them . You can just repeat the array expression and call the methods on that, but that is inelegant. All FAR++ lazyArrays have a `stablise` method that allows them to be stored and used later. Because lazyArrays have very complex types you should always capture a stabilised lazyArray into an auto typed variable. The below example shows how you can use this to copy values from an array expression into an `std::vector` using `std::copy`.

```
#include <iostream>
#include <vector>
#include "far.h"

int main(){

    far::Array<double,2> a = far::reshape(far::linspace(0.0,2.0,5*4),5,4);
    auto mod = far::sin(a+1).stabilise();

    std::vector<double> b;
    std::copy(mod.begin(),mod.end(),std::back_inserter(b));
    std::cout << "Elements from the vector are: ";
    for (auto element : mod){
        std::cout << element << " ";
    }
    std::cout << "\n";

    std::cout << "Elements from the array are: " << far::sin(a+1) << "\n";
}
```

You can also get data from an STL container easily into a FAR++ array in various ways. There is an overload to `toArray` that takes any container that implements begin, end and size and converts it into a rank 1 FAR++ array holding the same values. As well as using `toArray` you can assign to a FAR++ array using normal C++ mechanisms, such as `std::copy`, iterator copying or just using array indices. FAR++ arrays do not have inserter iterators because they are not a growing container and must be

allocated to the correct size before data can be copied into them.

```
#include <iostream>
#include <vector>
#include <list>
#include <deque>
#include "far.h"

int main(){
    std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    far::Array<int,1> a = far::toArray(v); //Convert iterable to far::Array
    std::cout << "Vector to FAR++ rank 1 array\n" << a << "\n\n";

    std::list<int> l = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    far::Array<int,2> b = far::reshape(far::toArray(l),3,3);
    std::cout << "List to FAR++ rank 2 array\n";
    std::cout << far::gridPrint(b) << "\n\n";

    std::deque<int> d = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    far::Array<int,2> c(3,3);
    auto cIter = c.begin();
    for(auto &val : d){
        *cIter = val;
        cIter++;
    }
    std::cout << "Deque to FAR++ rank 2 array using iterator assignment to
    ↪  array\n";
    std::cout << far::gridPrint(c) << "\n\n";

    c=0;
    std::copy(v.begin(), v.end(), c.begin());
    std::cout << "Vector to FAR++ rank 2 array using std::copy\n";
    std::cout << far::gridPrint(c) << "\n";

}
```

To provide a more native C++ feel to FAR++, FAR++ arrays have methods for testing for sizes etc. as well as the Fortran function-based inquiry mechanisms. These methods are

- size_t size() - Get the total number of elements in the array
- size_t rankSize(int rank) - Get the total number of elements in direction rank, with rank starting from 1
- int64_t LB(int rank) - Get the lower bound of direction rank, with rank starting from 1

- int64_t UB(int rank) - Get the upper bound of direction rank, with rank starting from 1
- bool contiguous() - Is this array's data contiguous in memory
- bool allocated() - Is this array allocated (i.e. has memory associated with it)
- bool associated() - Is this array a pointer pointing to another array
- bool associated(T_other &other) - Is this array a pointer pointing to other or the same target as other if other is a pointer

#### Elemental functions

One useful feature of Fortran is that you can write functions that work both on simple values and apply element-wise to arrays. These functions are called "elemental functions", and they cannot be directly done in C++, but we have created a method for doing something similar. There are two parts

- Create a safe mechanism by which a function can take either a value or an array of values
- Then process these values in a specified way

The first is solved by providing suitable concepts and the second is provided by wrapping the same internal mechanisms that FAR++ uses in a more convenient front end. It is important to note that in order for these to work efficiently you have to make proper use of const qualifiers.

```
template<far::elementalParameter<double> T_x,
 ↪  far::elementalParameter<double> T_mean, far::elementalParameter<double>
 ↪  T_stddev>
auto gaussian(const T_x &x, const T_mean &mean, const T_stddev &stdev){
    using T_x_inner = far::arrayInfo_t<T_x>;
    using T_mean_inner = far::arrayInfo_t<T_mean>;
    using T_stddev_inner = far::arrayInfo_t<T_stddev>;
    auto l = [](const T_x_inner &x, const T_mean_inner &mean, const
     ↪  T_stddev_inner &stdev){
        return exp(-(x-mean)*(x-mean)/(2*stdev*stdev));
    };
    return far::makeElemental<l>(std::forward<const T_x &>(x),
     ↪  std::forward<const T_mean&>(mean), std::forward<const
     ↪  T_stddev&>(stdev));
}
```

The above code snippet shows how to create an elemental function to calculate a Gaussian function, with the mean, the standard deviation and the value to evaluate it at all being possibly arrays. Breaking this down you can see the different elements

`far::elementalParameter<double>` is a concept that describes that idea of either a FAR++ array, FAR++ contiguous array, FAR++ lazyArray or a simple value having the specified type. Any rank of

array, contiguous array or lazyArray will match the concept. This is used to create the parameters to this function, which must use the template concepts, but otherwise can be values, references, const references, universal references etc. as wanted.

To create the function that does the actual work (the inner function) you need to know the type of the elements of any arrays that are passed to the function. You can do that with the `far::arrayInfo_t<T_x>` type alias. If the type passed to it is a non array type then arrayInfo will return that type unchanged, if it is a FAR++ array then it will return the type of the elements of the array, so the `using` statements above return either the type of the parameters (if they are not arrays), or the type of the elements of the array (if they are arrays).

With those types created we then create a lambda that implements the function that will evaluate the Gaussian function for single elements. All of the parameters to this function are const references, and it is important that they are either const references or values as will be described later. This function does not have to be a lambda, it can be a free function but it cannot currently be a functor. If it is a lambda then it must have no captures. If it is a free function then it must be unambiguously resolvable at compile time by name, and must take parameters that can be fulfilled by the core type of any array parameters and the type of any non-array parameters. Type conversion is performed as normal, so value parameters to the inner function have to be possible to convert to, while reference parameters have to match exactly. The return type of the inner function is the core return type of the elemental function, although the detailed return type will either be a value of this type, an Array of this type or a lazyArray of this type. If all parameters are non-array parameters then the result will be a value of the type of the return from the inner function. If any parameters are array parameters then the result will be an array or lazyArray having the same rank as the array parameters to the elemental function and the type of the return from the inner function. This variation in type means that it is easiest if the outer function has `auto` return type.

Finally, we use the function `far::makeElemental` to create the actual elemental function. This function is templated on the function that acts on the individual elements and takes the parameters that we passed to the encapsulating function. Note that correct behaviour under all circumstances requires perfect forwarding of the arguments to the outer function to `makeElemental`. If you are not familiar with this then it requires that you write `std::forward<typespec>(parameter)` for each parameter, where `typespec` must match exactly the type of the parameter to the outer function. So here, we have `std::forward<const T_x&>(x)` which correctly ensures that the `makeElemental` function sees the parameter as being a constant reference to `x`. You have to do this perfect forwarding even if the parameter is just a simple value, so `std::forward<int>(a)` would forward an integer value a to `makeElemental`. If you forget to do this forwarding correctly, things will mostly work as expected so long as you stick to reference and const reference variables, but parameters passed by value will be incorrectly stored as references and then fail to work properly. If you find yourself getting weird random errors in elemental functions, check that you have used

`std::forward` for *all* the parameters to `makeElemental`

The function `gaussian` is now ready and can be called with any combination of arrays, array expressions and values.

```
    std::cout << "Gaussian function works on single values\n";
    std::cout << "Gaussian(1, 0, 1) = " << gaussian(1.0, 0.0, 1.0) << "\n";
    std::cout << "Gaussian(0, 0, 1) = " << gaussian(0.0, 0.0, 1.0) << "\n";
    std::cout << "Gaussian(-1, 0, 1) = " << gaussian(-1.0, 0.0, 1.0) <<
    ↪  "\n";

    std::cout << "\n\n";

    far::Array<double,1> x = {-2,-1,0,1,2};
    far::Array<double,1> mean = {0,0,0,0,0};
    far::Array<double,1> stdev = {1,1,1,1,1};
    std::cout << "Gaussian function works when all parameters are
    ↪  arrays\n";
    std::cout << "Gaussian([-2,-1,0,1,2], [0,0,0,0,0], [1,1,1,1,1]) = " <<
    ↪  gaussian(x, mean, stdev) << "\n";
```

Gaussian returns a lazyArray. This is automatically chosen by FAR++ because the parameters to the inner function are const references and the function returns a value. This means that it assumes that the only result of evaluating the expression is that a calculated value is returned by the function. This is not true if the function has side effects, but this case will be covered later. LazyArrays have substantial performance and memory advantages over creating array temporaries, so you should always flag parameters to inner functions as const (or simple values) if their values will not be changed by the function to allow this optimisation.

Sometimes, one wants to have elemental subroutines (as they would be called in Fortran). That is functions that modify their arguments instead of returning a calulated value. This is done in exactly the same way, but the inner function does not flag its parameters as const.

```
template<far::elementalParameter<double> T_x>
void doubler(T_x &x){
    using T_x_inner = far::arrayInfo_t<T_x>;
    auto l = [](T_x_inner &x){
         x*=2;
    };
    far::makeElemental<l>(std::forward<T_x&>(x));
}
```

This function again can be called on either values (although, of course not literal values now) or arrays again. If you try to call such a function with an array expression, it will fail because doubler itself cannot

accept an rvalue. When this function is called, it is evaluated immediately and the value or array that is passed to it is modified immediately.

It is, of course, possible to have a function that modifies its arguments as well as returning a value. By default this will be evaluated like the subroutine - the parameters will be immediately modified and an array (an actual `Array`, not a `lazyArray`) will be returned containing the results.

```
template<far::elementalParameter<double> T_x>
auto doubler(T_x &x){
    using T_x_inner = far::arrayInfo_t<T_x>;
    auto l = [](T_x_inner &x){
        T_x_inner y=x;
         x*=2;
        return y;
    };
    return far::makeElemental<l>(std::forward<T_x&>(x));
}
```

If you use such an elemental function in an array expression it will work, but performance may be substantially lower than the version that returns a `lazyArray`, so make careful use of `const`. It is not possible within C++ to detect if a function is truly pure, and the use of a lazyArray might produce unexpected results if you modify global state or call external library calls from within your code.

For a simple example, let us consider the simplest possible side effect - printing to screen.

```
#include <iostream>
#include "far.h"

template<far::elementalParameter<double> T_x>
auto doubler(const T_x &x){
    using T_x_inner = far::arrayInfo_t<T_x>;
    auto l = [](const T_x_inner &x){
        std::cout << "Using value " << x << std::endl;
        return x*2;
    };
    return far::makeElemental<l>(std::forward<const T_x&>(x));
}

template<far::elementalParameter<double> T_x>
auto alldoubler(T_x &x){
    using T_x_inner = far::arrayInfo_t<T_x>;
    auto l = [](T_x_inner &x){
        std::cout << "Using value " << x << std::endl;
        return x*2;
```

```
    };
    return far::makeElemental<l>(std::forward<T_x&>(x));
}

int main(){

    far::Array<double,1> a = far::linspace<double>(1,10,10);
    far::Array<double,1> b = a;
    far::Array<double,1> c = far::linspace<double>(-4,5,10);

    std::cout << "Testing lazy doubler\n";
    auto r1 = far::toArray(far::merge(a,doubler(b),c>0));
    std::cout << "\nTesting array doubler\n";
    auto r2 = far::toArray(far::merge(a,alldoubler(b),c>0));

    std::cout << "Result 1 is " << r1 << "\n";
    std::cout << "Result 2 is " << r2 << "\n";
}
```

The elemental functions include a print statement so that they print the value of the element being evaluated, which because of how the array is set up is also the element number of the element being acted on. The `merge` function is then used to use only some of the elements of the result of each of the functions. The basic doubler function only prints the values of the elements that are being used by merge, whereas the alldoubler function, that is different only in that its parameter is not const, generates all elements of the array that it returns so all of the elements are printed. This shows the problem with elemental functions with side effects - if they are evaluated in lazy form then they only cause side effects for the elements that are ever used by other parts of the code.

As a consequence of this, it is possible to pass a flag to makeElemental that will force it to be either a lazyArray returning elemental function or an Array returning elemental function. This flag is the second template parameter to makeElemental and can have the following values

- far::ert::Auto - Automatic - the default behaviour
- far::ert::Lazy - Will always return a lazy array
- far::ert::Array - Will always return an array

```
#include <iostream>
#include "far.h"

template<far::elementalParameter<double> T_x>
auto doubler(const T_x &x){
    using T_x_inner = far::arrayInfo_t<T_x>;
    auto l = [](const T_x_inner &x){
```

```
        std::cout << "Using value " << x << std::endl;
        return x*2;
    };
    return far::makeElemental<l,far::ert::Array>(std::forward<const
    ↪  T_x&>(x));
}

template<far::elementalParameter<double> T_x>
auto alldoubler(const T_x &x){
    using T_x_inner = far::arrayInfo_t<T_x>;
    auto l = [](T_x_inner &x){
        std::cout << "Using value " << x << std::endl;
        return x*2;
    };
    return far::makeElemental<l,far::ert::Lazy>(std::forward<const
    ↪  T_x&>(x));
}

int main(){

    far::Array<double,1> a = far::linspace<double>(1,10,10);
    far::Array<double,1> b = a;
    far::Array<double,1> c = far::linspace<double>(-4,5,10);

    std::cout << "Testing naturally lazy overriden to array doubler\n";
    auto r1 = far::toArray(far::merge(a,doubler(b),c>0));
    std::cout << "\nTesting naturally array overriden to lazy doubler\n";
    auto r2 = far::toArray(far::merge(a,alldoubler(b),c>0));

    std::cout << "Result 1 is " << r1 << "\n";
    std::cout << "Result 2 is " << r2 << "\n";
}
```

Running this, you will see that the behaviour of the functions is reversed. Also, note that the parameters to `std::forward` match the *outer* function parameters, not the *inner* function parameters. For `alldoubler` the inner function has non-const parameters, but because the outer function has a const parameter the parameter to `std::forward` *must* be const. Generally, you should only do this kind of override in two cases

1) You should override to `Array` if you know that your elemental function has side effects but you want to have the parameters to your inner function be const since they are not modified
2) You should override to `Lazy` if you know that you will always be using all elements of the result

of your elemental function once and only once and you want the performance of a lazyArray

Otherwise, you should flag your inputs to your inner function as either const or non-const as appropriate and allow auto selection. Remember that if your function does not return any value then even if all of the parameters are const, it is assumed that there will be side effects (otherwise, what is such a function doing?) so an Array is created by default.

The above example will only work for `double` values and `double` arrays, and this matches the restrictions that Fortran places on elemental functions. Just as in Fortran you can overload the name of an elemental function for different types, and you do this by simply changing the type in the elementalParameter concept.

```
#include <iostream>
#include "far.h"

template<far::elementalParameter<double> T_x,
↪  far::elementalParameter<double> T_mean, far::elementalParameter<double>
↪  T_stddev>
auto gaussian(const T_x &x, const T_mean &mean, const T_stddev &stdev){
    using T_x_inner = far::arrayInfo_t<T_x>;
    using T_mean_inner = far::arrayInfo_t<T_mean>;
    using T_stddev_inner = far::arrayInfo_t<T_stddev>;
    std::cout << "Double version :";
    auto l = [](const T_x_inner &x, const T_mean_inner &mean, const
    ↪  T_stddev_inner &stdev){
        return exp(-(x-mean)*(x-mean)/(2*stdev*stdev));
    };
    return far::makeElemental<l>(std::forward<const T_x&>(x),
    ↪  std::forward<const T_mean&>(mean), std::forward<const
    ↪  T_stddev&>(stdev));
}

template<far::elementalParameter<float> T_x, far::elementalParameter<float>
↪  T_mean, far::elementalParameter<float> T_stddev>
auto gaussian(const T_x &x, const T_mean &mean, const T_stddev &stdev){
    using T_x_inner = far::arrayInfo_t<T_x>;
    using T_mean_inner = far::arrayInfo_t<T_mean>;
    using T_stddev_inner = far::arrayInfo_t<T_stddev>;
    std::cout << "Float version  :";
    auto l = [](const T_x_inner &x, const T_mean_inner &mean, const
    ↪  T_stddev_inner &stdev){
        return exp(-(x-mean)*(x-mean)/(2*stdev*stdev));
    };
```

```
    return far::makeElemental<l>(std::forward<const T_x&>(x),
    ↪  std::forward<const T_mean&>(mean), std::forward<const
    ↪  T_stddev&>(stdev));
}

int main(){

  std::cout << gaussian(1.0,0.0,1.0) << "\n";
  std::cout << gaussian(1.0f,0.0f,1.0f) << "\n";

}
```

This is necessary if you want different types to have different behaviours, but there are often cases where the behaviour of the different types is the same, as demonstrated above for float and double. Furthermore, while it is fairly easy to create a float version and a double version, what about if you want to have a version where any parameter can freely be a float or a double, or array thereof? C++ allows us more flexibility than Fortran here. elementalParameter doesn't just allow a single type parameter, but allows you to specify a comma separated list of parameters, and matching any of them is sufficient. Note that doing this can potentially cause ambiguity issues if combined with overloads - care must be taken.

```
#include <iostream>
#include "far.h"

template<far::elementalParameter<double,float> T_x,
↪  far::elementalParameter<double,float> T_mean,
↪  far::elementalParameter<double,float> T_stddev>
auto gaussian(const T_x &x, const T_mean &mean, const T_stddev &stdev){
    using T_x_inner = far::arrayInfo_t<T_x>;
    using T_mean_inner = far::arrayInfo_t<T_mean>;
    using T_stddev_inner = far::arrayInfo_t<T_stddev>;
    auto l = [](const T_x_inner &x, const T_mean_inner &mean, const
    ↪  T_stddev_inner &stdev){
        return exp(-(x-mean)*(x-mean)/(2*stdev*stdev));
    };
    return far::makeElemental<l>(std::forward<const T_x&>(x),
    ↪  std::forward<const T_mean&>(mean), std::forward<const
    ↪  T_stddev&>(stdev));
}

int main(){

  std::cout << gaussian(1.0,0.0,1.0) << "\n";
```

```
  std::cout << gaussian(1.0f,0.0f,1.0f) << "\n";
  std::cout << gaussian(1.0,0.0f,1.0) << "\n";

}
```

This version matches any or all parameters being of double or float type. You can have as many types in elementalParameter as you like, but obviously you must be careful to ensure that all possible combinations of types is valid.

Finally, you can have array parameters to elemental functions that are arrays but are not accessed elementally, that is they are still arrays when they are passed to the inner function. Doing this is very simple, just create an array parameter to the inner function instead of a parameter matching the elements of an array and change the concept used on the outer function from `elementalParameter` which will match either a value or an array for another FAR++ concept (detailed later in full) to require an array parameter (usually `arrayParameter` or `manyTypeArrayParameter`).

```
#include <iostream>
#include "far.h"


template<far::elementalParameter<std::integral> T1,
 ↪  far::arrayParameter<double,1> T2>
auto lookup(const T1& index, T2& data){

  using T1_inner = far::arrayInfo_t<T1>;
  auto l = []( const T1_inner & index, T2 &data){return data(index);};
  return makeElemental<l>(std::forward<const T1&>(index),
   ↪  std::forward<T2&>(data));

}

int main(){

  far::Array<double,1> A = far::linspace(11,20,10);
  std::cout << lookup(1,A) << "\n";
  std::cout << lookup(far::linspace<int>(10,1,10),A) << "\n";

}
```

This simple example shows using a 1D integer array as a lookup table and allows you to pass either a value to have it looked up or an array of indices, in which case the result is an array of the looked up values. Of minor note is the use of `std::integral` which is a built in concept corresponding to any integer type, allowing this elemental function to easily be called with any integer.

A final note on elemental functions - how they choose the bounds over which to run the function. It is not valid for any parameters that are arrays accessed elementally to be a different rank or shape to each other, but how exactly is the rank and size selected? The answer is fairly simple - the highest rank parameter to the function that is an array that is not an array parameter to the inner function is used to determine the rank of the array, and the first parameter of that rank that is not an array parameter to the inner function is used to determine the range over which the function runs. The arrays don't have to have the same bounds, just the same total number of elements in each direction.

**Concepts**

We have already seen how useful C++20 concepts can be when using FAR++, but there are many concepts that have not been described. To describe them, their name will be given followed by any template parameters they take in <>, followed by a description of what they do. If they take no template parameters they will be followed with <>. You do not need to use empty angle brackets if using these concepts. If they take optionally multiple parameters then the parameter will be followed with …

- anyArray<> - Matches any array or lazyArray of any rank or type
- notArray<> - Matches anything that is not an array or lazyArray
- anyRealArray<> - Matches any floating point (i.e. float, double, long double) array of any rank
- anyIntegerArray<> - Matches any integer array (i.e. int8_t, int16_t, int32_t, int64_t and larger types is supported as well as unsigned integers) of any rank
- anyRankArray<typename…> - Matches an array of any rank with a type matching one of a comma separated list of types
- anyRankConvertibleArray<typename…> - Matches an array of any rank with a type that can be converted to any of a comma separated list of types
- anyTypeArray - Matches an array of any type with the specified rank
- arrayParameter<typename,int rank> - Matches an array having a specific type and rank
- manyTypeArrayParameter<int rank, typename…> - Matches an array having a specific rank and a type from a comma separated list of types
- elementalParameter<typename…> - Matches any array or non array value matching a specified type from a comma separated list of types
- rankIntegerArray - Matches any integer array (i.e. int8_t, int16_t, int32_t, int64_t and larger types is supported as well as unsigned integers) of the specified rank
- rankRealArray - Matches any floating point (i.e. float, double, long double) array of the specified rank

Using these concepts should make it much easier to pass FAR++ arrays around your code. FAR++ arrays will generally deep copy when passed to a function as a value, so you should always pass them to functions as a reference.

### Interoperability with C

We have already seen that FAR++ arrays are able to interact with normal C++ using their random access iterators, but sometimes it is useful to be able to use a FAR++ array in a context where you would normally use a C array or pointer. A good example of this would be to use a library written in C, such as FFTW. The easiest way of doing this is to use the `data` method, just as you would for a vector.

```
#include <iostream>
#include "far.h"

int main(){

    far::Array<int,2> a = reshape(far::linspace(1, 9, 9),3,3);
    int *data = a.data();
    for(int i=0; i<a.size(); i++){
        std::cout << data[i] << " ";
    }
    std::cout << std::endl;

}
```

So long as your array is contiguous, data will work exactly the same as it does for a vector, but unlike a vector a FAR++ array doesn't have to be contiguous. There is no trivial way of accessing the data using a pointer if the underlying data is not contiguous. If you try to use the data method on a non-contiguous array it will cause a `std::runtime_error` to be raised (the precompiler directive `FAR_BAD_DATA_IS_NULL` can be set to switch this to returning a NULL pointer). The same behaviour occurs if you call the data method on a lazyArray (i.e. on an array expression). These limit the use of the data method, but it is the easiest way of accessing the underlying memory of an Array. There is a more general approach that works for non-contiguous arrays and for lazyArrays. This uses a trio of functions called `makeCopyInOut`, `makeCopyIn`, `makeCopyOut`.

It is quite easy to make non-contiguous arrays or lazyArrays have a contiguous pointer equivalent - you simply copy them into contiguous memory. This has a two problems - it uses memory and it breaks the model for how FAR++ arrays work if you are not careful. The first is unavoidable - if you want data in contiguous memory and it is not in contiguous memory then you have to copy it, but the second is annoying and at least partially fixable. The problem is that you normally expect that any access that you have on FAR++ arrays is a view onto the underlying data. So slicing an array doesn't stop you from assigning values to it. Once you are working with a copy of the original memory this link is broken, but it can be partly restored by having data copied back into the original once the copy goes out of scope. In FAR++ this is achieved by another class called `contiguous`. The `contiguous` class is not itself an array, or a C style pointer to the underlying data, but it has casting operators to both, so it can be used

anywhere that either can be used so long as the types are unambiguous. You don't manually create `contiguous` objects, but create them using `makeCopyInOut`, `makeCopyIn` and `makeCopyOut`. The names suggest what they do. `makeCopyInOut` creates a contiguous version of the data in an array or expression, copies the initial state of the source into the array and then copies any changes back to the original array when the `contiguous` object goes out of scope. `makeCopyIn` skips the final copying out stage and is used when you want access to the data in an array or array expression as a pointer but don't intend to change it. A good example of a function that would work like that is `MPI_Send`. `makeCopyIn` is the only one of these functions that will work properly when used with a lazyArray/array expression because copying back data to a lazyArray is normally not meaningful. `makeCopyOut` is the opposite again - the memory that the pointer points to is the same size as the array that you are making a contiguous copy of, but it is not populated with the initial state of the original array. When the `contiguous` object goes out of scope the data that is written to it is copied back to the original array. This could be used with a function like `MPI_Recv`.

If the array that you are making contiguous is already contiguous (including runtime contiguous, so a slice of an array that happens to be contiguous would qualify) then the `contiguous` object becomes a thin wrapper around the underlying memory and the copying phases are avoided, so DO NOT RELY on being able to change the data behind a pointer without affecting the original data just because you specified `makeCopyIn`.

```
#include <iostream>
#include "far.h"

void ptrFunc(int *data, int size){
    std::cout << "Data from pointer : ";
    for(int i=0; i<size; i++){
        std::cout << data[i] << " ";
    }
    std::cout << std::endl;
}

void arrayFunc(const far::Array<int,2> &a){
    std::cout << "Data from array from contiguous object : ";
    std::cout << a << std::endl;
}

int main(){

    far::Array<int,2> a = reshape(far::linspace(1, 9, 9),3,3);
    //Print both using pointer and using array
    ptrFunc(far::makeCopyIn(a), 9);
    arrayFunc(far::makeCopyIn(a));
```

```
    //Now for a range
    ptrFunc(far::makeCopyIn(a(far::Range(2,3),far::Range(1,3))), 6);
    arrayFunc(far::makeCopyIn(a(far::Range(2,3),far::Range(1,3))));

    //Now for an expression. Note that this has to be makeCopyIn because
    ↪  writing to an expression
    //isn't ever valid
    ptrFunc(far::makeCopyIn(a*2), 9);
    arrayFunc(far::makeCopyIn(a*2));

}
```

This simple example shows you accessing both contiguous (a) and non-contiguous (a(`far::Range(2,3),far::Ra` and a*2) arrays using both pointers and arrays. Note that in both cases, the type of the parameter that the result of `makeCopy*` is being assigned to must be unambiguous. Because this behaviour relies on a casting operator on the `contiguous` object it can only be passed to functions that expect either a pointer to the core datatype of the array that was passed to `makeCopy*` or to an array of the same type and rank as the array passed to `makeCopy*`.

You have various options to ensure that that happens. The easiest is just to make sure that the function that you are calling has the same specific type as your array and the casting operator will be automatic. Alternatively, you can select a specific version of a template function, or explicitly cast your `contiguous` object to the type of a pointer to the underlying array data. If all else fails, there is a `toPointer` method of the `contiguous` class which will return a pointer to the underlying data so that normal type resolution will occur. All of these options are shown below

```
#include <iostream>
#include "far.h"

template<typename T>
void ptrFunc(T *data, int size){
    std::cout << "Data from pointer : ";
    for(int i=0; i<size; i++){
        std::cout << data[i] << " ";
    }
    std::cout << std::endl;
}

int main(){

    far::Array<int,2> a = reshape(far::linspace(1, 9, 9),3,3);
```

```
    //Can make the type unambiguous by specifying the template type
    ptrFunc<int>(far::makeCopyIn(a(far::Range(2,3),far::Range(1,3))), 6);

    //Can make the type unambiguous by casting
    ptr-
↪   Func(static_cast<int*>(far::makeCopyIn(a(far::Range(2,3),far::Range(1,3)))),6);

    //Can make unambiguous by using toPointer
    ptr-
↪   Func(far::makeCopyIn(a(far::Range(2,3),far::Range(1,3))).toPointer(),6);
}
```

makeCopyOut is used in much the same way, but cannot be used on lazyArrays/array expressions.

```
#include <iostream>
#include "far.h"

void ptrFunc(int *data, int size){
    for(int i=0; i<size; i++){
        data[i]=i;
    }
}

int main(){

    far::Array<int,2> a(3,3);
    a=-1; //Set all elements to -1 so that you can see all changed elements
    ptrFunc(far::makeCopyOut(a), 9); //Alter the whole array
    std::cout << far::gridPrint(a) << std::endl;
    a=-1; //Set back to -1
    ptrFunc(far::makeCopyOut(a(far::Range(2,3),far::Range(1,3))), 6);
↪   //Alter only a non-contiguous range
    std::cout << far::gridPrint(a) << std::endl;
}
```

In this simple example you assign elements of a FAR++ array using a C style array, showing how makeCopyOut works. makeCopyInOut works in basically the same way but combines makeCopyIn and makeCopyOut.

```
#include <iostream>
#include "far.h"

void ptrFunc(int *data, int size){
    for(int i=0; i<size; i++){
```

```
        data[i]*=-2;
    }
}

int main(){

    far::Array<int,2> a = reshape(far::linspace(1, 9, 9),3,3);
    ptrFunc(far::makeCopyInOut(a), 9); //Alter the whole array
    std::cout << far::gridPrint(a) << std::endl;
    a=reshape(far::linspace(1, 9, 9),3,3);
    ptrFunc(far::makeCopyInOut(a(far::Range(2,3),far::Range(1,3))), 6);
↪   //Alter only a non-contiguous range
    std::cout << far::gridPrint(a) << std::endl;
}
```

This code iterates over the elements of either an array or an array subsection and multiplies every element by -2. Printing the resulting array shows that any initial value assigned to an element is modified as specified. To a great degree, this is all there is to `contiguous` as it is supposed to be used. You use it to allow you to pass FAR++ arrays to a function that expects a C pointer.

There are some caveats. Note that the while the `copyIn` behaviour occurs when the `contiguous` type is created, the `copyOut` only triggers when the `contiguous` object goes out of scope. That is not the same as saying that the data is only updated when the `contiguous` object goes out of scope, because if the original array's memory is already contiguous then no copy is made and you work with the original memory, so no copy back is needed and the changes are immediate. These effects can be seen as follows

```
#include <iostream>
#include "far.h"

void ptrFunc(int *data, int size){
    for(int i=0; i<size; i++){
        data[i]=i;
    }
}

int main(){

    far::Array<int,2> a(3,3);
    {
        a=-1; //Set all elements to -1 so that you can see all changed
↪   elements
        auto co = far::makeCopyOut(a); //Create a copy out object
```

```
        ptrFunc(co, 9); //Alter the whole array
        std::cout << "I have updated the whole array, so even though
        ↪  CopyOut object is in scope update has happened\n";
        std::cout << far::gridPrint(a) << std::endl; // This will have
        ↪  already updated
    }
    {
        a=-1; //Set back to -1
        auto co = far::makeCopyOut(a(far::Range(2,3),far::Range(1,3)));
        ↪  //Create a copy out object for a non-contiguous range
        ptrFunc(co, 6); //Alter only a non-contiguous range
        std::cout << "I have updated only a non-contiguous range, so since
        ↪  CopyOut object is in scope, no update has happened\n";
        std::cout << far::gridPrint(a) << std::endl;
    }
    std::cout << "CopyOut object has gone out of scope, so the update will
    ↪  happen now\n";
    std::cout << far::gridPrint(a) << std::endl;
}
```

This code shows what is described above. The array automatically updates as soon as changes are made because the `contiguous` object is a thin wrapper around a whole array or contiguous array section. The non-contiguous array slice will not update until the `contiguous` object goes out of scope. You can manually cause copies to happen using the `copyBack` method of `contiguous` object. When the `contiguous` object goes out of scope another copy will happen because it is assumed that the only reason for wanting to keep a `contiguous` object alive is that you want to make further changes to it. Should you wish to avoid this but still keep a `contiguous` object around as a variable, there is an `unbind` method that acts as if the `contiguous` object went out of scope. The data is copied back to the source array and the `contiguous` object is unlinked from that array. You cannot now use the `contiguous` object for any reason.

#### FAR++ Fortran interoperability

One of the obvious uses for a library which allows Fortran like access to arrays is to actually interact with Fortran code. Essentially in order to make this possible one has to bind memory that is not controlled by FAR++ to an array object. That can be done with the `bind` method. This method takes a pointer to the memory to be bound to the array and a set of range specifiers matching the rank of the array. Any range specifier is permitted, either a simple value or a `Range` object.

```
#include <iostream>
#include <vector>
```

```cpp
#include "far.h"

int main(){

    std::vector<int> v;
    v.resize(9);
    for(int i=0; i<9; i++){
        v[i] = i+1;
    }

    far::Array<int,2> a;
    //Bind the data of the vector to the array
    a.bind(v.data(), 3,3);
    std::cout << "You can now access the vector elements as an array\n";
    std::cout << "Array" << std::endl;
    std::cout << far::gridPrint(a) << std::endl;
    std::cout << "Array after adding 1 and doubling" << std::endl;
    std::cout << far::gridPrint((a+1)*2) << std::endl;

    a=far::reshape(far::linspace(9,1,9),3,3);
    std::cout << "Changing the array changes the vector" << std::endl;
    for (auto &el:v){
        std::cout << el << " ";
    }
    std::cout << "\n";

}
```

When you bind an external pointer to a FAR++ array the array takes no responsibility for the deleting the memory. In this case, that is correct because the vector is responsible for the lifetime of it's own internal memory, but in theory this can cause a memory leak if you lose access to the memory that you have bound to a FAR++ array. It is important that for any memory that you give to FAR++ you have another record of it so that it can be deallocated. Bound FAR++ arrays behave like pointer arrays and array slices, so they do not automatically reallocate when another array is assigned to them. Generally this behaviour is what would be wanted because if an array is pointing to an external memory representation you do not want that link to be severed accidentally. Deallocating the array will break the link to the remote memory. As an alternative you may use the `takeOwnership` method. This works much like the `bind` method, but the FAR++ array takes ownership of the memory and deletes it when needed. Arrays that take ownership of memory will deallocate an reallocate as normal arrays. *Be very careful with this function. If the memory was not allocated with new it may not deallocate correctly*. We do not recommend using `takeOwnership` except in very limited circumstances.

Interacting with Fortran is fairly simple. You just need to use Fortran/C interoperability to pass a suitable pointer in the correct direction. So accessing memory allocated in Fortran from C would be

```
MODULE test
    USE, INTRINSIC :: ISO_C_BINDING
    IMPLICIT NONE
    REAL(C_FLOAT), DIMENSION(:,:), TARGET,ALLOCATABLE :: A
    CONTAINS
    FUNCTION getA(nx,ny) BIND(C,name="getA") !Bind C with name to get
    ↪  capitalisation
        INTEGER(C_INT), VALUE :: nx,ny
        TYPE(C_PTR) :: getA
        INTEGER :: i, j
        IF (ALLOCATED(A)) DEALLOCATE(A)
        ALLOCATE(A(nx,ny))
        DO j=1,ny
            DO i=1,nx
                A(i,j) = (i-1)+(j-1)*nx + 1
            END DO
        END DO
        getA = C_LOC(A)
    END FUNCTION getA

    SUBROUTINE printA() BIND(C,name="printA")
        PRINT *, 'A:', A
    END SUBROUTINE printA
END MODULE test
```

This Fortran module creates two C interoperable functions. One to allocate an allocatable array to a specified size and return a C pointer to its data and one to print that array. That is then matched with a C++ program

```
#include <iostream>
#include "far.h"

extern "C" float* getA(int nx,int ny);
extern "C" void printA();

int main(){
    int nx=5,ny=5;
    float *a = getA(nx,ny);
    far::Array<float,2> A;
    A.bind(a,nx,ny);
    std::cout << "Array" << std::endl;
```

```
    std::cout << far::gridPrint(A) << std::endl;
    std::cout << "Array slice" << std::endl;
    std::cout << far::gridPrint(A(far::Range(2,3),far::Range(3,5))) <<
    ↪  std::endl;
    std::cout << "\n";

    //Now set it so that the values are reversed (i.e. nx*ny, nx*ny-1, ...)
    A = far::reshape(far::linspace(nx*ny,1,nx*ny),nx,ny);
    std::cout << "And printing from Fortran" << std::endl;
    printA();
}
```

Once you have bound the pointer to the underlying memory from Fortran to the FAR++ array, you can access it in C++ exactly as if it was a normal FAR++ array, and the indices and ordering match up between Fortran and C++. You can go in the other direction as well

```
#include <iostream>
#include "far.h"

far::Array<int,2> A;

extern "C" int* createArray(int nx,int ny);
extern "C" void printArray();

int* createArray(int nx,int ny){
    A = reshape(far::linspace(nx*ny,1,nx*ny),nx,ny);
    return A.data();
}

void printArray(){
    std::cout << "From C++" << std::endl;
    std::cout << far::gridPrint(A) << std::endl;
}
```

```
MODULE linkmod

    USE ISO_C_BINDING
    IMPLICIT NONE

    INTERFACE
        FUNCTION createArray(nx,ny) BIND(C,name="createArray")
            USE ISO_C_BINDING
            IMPLICIT NONE
            INTEGER(C_INT), VALUE :: nx,ny
```

```
            TYPE(C_PTR) :: createArray
        END FUNCTION createArray

        SUBROUTINE printArray() BIND(C,name="printArray")
            USE ISO_C_BINDING
            IMPLICIT NONE
        END SUBROUTINE printArray
    END INTERFACE

END MODULE linkmod

PROGRAM test
    USE linkmod
    IMPLICIT NONE

    INTEGER :: nx, ny, i
    TYPE(C_PTR) :: arrayPtr
    INTEGER(C_INT), POINTER, DIMENSION(:,:) :: A

    nx = 3
    ny = 4

    arrayPtr = createArray(nx,ny)
    CALL C_F_POINTER(arrayPtr, A, [nx,ny])
    PRINT *, "In Fortran array is ", A
    PRINT *, "Reversing order and printing from C++"
    A(:,:) = RESHAPE([(i,i=1,nx*ny)], [nx,ny])
    CALL printArray()


END PROGRAM test
```

This does almost exactly the same in the other direction. You have to be careful with making sure that the right language deals with allocating and deallocating the arrays (in particular FAR++ must *never* take ownership of memory allocated by Fortran), but otherwise, working with data between the languages is easy. Since interoperability with C is common in almost all languages this same approach can be used to interoperate with any other language. The only restriction comes from the fact that different languages use different approaches to laying out multidimensional arrays in memory. As you might be able to see here, FAR++ uses the same column major approach as Fortran and Matlab, but C/C++ and many other languages use row major layout, so how do we deal with this?

### Array majority

When dealing with multidimensional arrays you have to have some way of mapping the "true" underlying 1 dimensional strip of memory onto the axes of the multdimensional array. There are two general approaches [https://en.wikipedia.org/wiki/Row-_and_column-major_order], column major where the changes in the first index move between adjacent elements and row major where changes in the last index move between adjacent elements. It is worth pointing out that there are much more sophisticated approaches to laying out multidimensional arrays, but they are not covered here, and out of the box FAR++ doesn't support them.

Fortran is a column major language and so is FAR++ by default. If you simply want to change from column major to row major then simply define the precompiler flag `FAR_USE_C_INDEX`. *NOTE!* when you use precompiler directives with FAR++ you must define them before `far.h` is included *AND YOU MUST MAKE SURE THAT EVERY TIME YOU INCLUDE `far.h` YOU MUST HAVE THE SAME PRECOMPILER OPTIONS DEFINED*. You want to make sure that the view on `far.h` is consistent in every translation unit where it is included or strange and hard to diagnose errors can occur. The best solution is to add the precompiler directives to the compile line (normally using `-D` command line flags) rather than put `#define` instructions into your files. The latter is done here for simplicity, but it is not an ideal solution.

```
#include <iostream>
#define FAR_USE_C_INDEX
#include "far.h"

int main(){

   far::Array<int,2> a(3,3);

    std::cout << "Delta in x is " << &a(2,1) - &a(1,1) << std::endl;
    std::cout << "Delta in y is " << &a(1,2) - &a(1,1) << std::endl;

}
```

This example switches FAR++ to using C type row major ordering and then looks at the distances between adjacent elements in the two dimensions. The delta in x is 3, and the delta in y is 1, showing that the array is in row major order. Even though the array is now ordered like a C/C++ array the default lower bound is still 1 like in Fortran. You can modify the default lower bound by defining `FAR_DEFAULT_LB` to be the default lower bound.

```
#include <iostream>
#define FAR_DEFAULT_LB 0
#include "far.h"
```

```
int main(){

   far::Array<int,2> a = far::reshape(far::linspace(1,16,16),4,4);
   std::cout << far::gridPrint(a) << "\n";

}
```

This uses gridPrint to show that the arrays now have a zero lower bound. FAR_DEFAULT_LB can be used independently of FAR_USE_C_INDEX as shown above and gives you a Fortran ordered array with a default lower bound of 0. If you want to use both 0 based indexing and C ordering, so that FAR++ arrays work much like C++ arrays, you can define both of the previous precompiled directives, or just the single precompiler variable FAR_CSTYLE which sets both of them in a single directive.

Much as default index lower bounds can be overriden for individual arrays in FAR++, so can default ordering. This is done using the names CArray and FortranArray.

```
#include <iostream>
#include "far.h"

int main(){

    far::FortranArray<int,2> a = far::reshape(far::linspace(1,20,20),5,4);
    far::CArray<int,2> b = a;

    std::cout << "Fortran array\n";
    std::cout << far::gridPrint(a)<< "\n";
    std::cout << "In underlying order\n";
    for (auto e: a) std::cout << e << " ";
    std::cout << "\n\n";

    std::cout << "C array initialised by assigning from Fortran array\n";
    std::cout << far::gridPrint(b)<< "\n\n";
    std::cout << "In underlying order\n";
    for (auto e: b) std::cout << e << " ";
    std::cout << "\n\n";

}
```

When you run this code, you can see that the two arrays are the same, in the sense that elements in b with a given index have the same value as the same element in a, but in the underlying memory, they layout is completely different. a is column major, b is row major. This is the normal behaviour in FAR++. When arrays with different indexers interact, that interaction is not in the order of the underlying

memory, but based on the multi-dimensional index.

So, if I assigned the output from `reshape` to my `CArray` directly the numbers would be laid out in row major order in memory? Sadly, no as the following example shows

```
#include <iostream>
#include "far.h"

int main(){

    far::CArray<int,2> a = far::reshape(far::linspace(1,20,20),5,4);
    far::CArray<int,2> b =
↪   far::reshape(far::toCArray(far::linspace(1,20,20)),5,4);

    std::cout << "Simple assign from reshape(linspace)\n";
    std::cout << far::gridPrint(a)<< "\n";
    std::cout << "In underlying order\n";
    for (auto e: a) std::cout << e << " ";
    std::cout << "\n\n";

    std::cout << "Converting linspace to C array before reshape\n";
    std::cout << far::gridPrint(b)<< "\n\n";
    std::cout << "In underlying order\n";
    for (auto e: b) std::cout << e << " ";
    std::cout << "\n\n";

}
```

`reshape` has to choose some way of picking the ordering that it uses when reshaping an array and the solution that it uses is to use the same ordering as the array that is being reshaped. `linspace` produces arrays with the default ordering, here column major ordering, so it produces a column major ordered array. For a rank 1 array, it isn't really meaningful to talk about column or row major ordering, but the array that is produced still carries with it the idea that it is a column major ordered array. In theory, reshape could take a parameter to allow you to change the ordering, but it is much easier to just convert the output of `linspace` to be a row major array. This is done with the `toCArray` function. Note that this function constructs a C array with the exact same values at the same indices as the original array, it does *not* produce a row major array with the underlying data in the same order. To further clarify this, `toCArray` and `toFortranArray` change the way the data is laid out in memory, but do not change the position of values in the array. This means that applying `toCArray` to the result of reshape will do nothing that assignment to a `CArray` variable wouldn't do anyway. Applying `toCArray` to the result of `linspace` changes the 1D array to be a row major array that `reshape` then reshapes into a rank 2 array.

It is recommended that you do not write code that mixes `CArray` and `FortranArray` types. Just use one or the other, switching using the precompiler flags. The main reason to need both is if you are working both with a C library and a Fortran library that each expect their native type of array ordering and then you can easily have the same data prepared in both ways.

#### Advanced topics in elemental functions (ALPHA FEATURE)

Elemental functions as described above work much as in Fortran, but unlike Fortran we have access to more of the internal ways in which elemental functions work and that gives us more options. Note that everything in this section is an alpha feature and may change or even be removed if it is necessary to allow a core feature to work.

The main thing that permits more sophisticated use of elemental functions actually makes them partly non-elemental. You can add another parameter of type `indexInfo<rank>` anywhere in the parameter list of the inner function and it will be contain the information about which element of the array is being worked on. Now, rather than putting parameters to the inner function that match an element of the parameters, you should put the type of the parameter to the outer function and you can access it. There is now a function called `getItem(array,indexInfo)` that you can use to get the element of the array that is currently being considered.

```
#include <iostream>
#include "far.h"

template<far::elementalParameter<double> T_x>
auto doubler(T_x &x){
    using T_x_inner = far::arrayInfo_t<T_x>;
    constexpr int rank = far::arrayInfo<T_x>::rank;
    auto l = [](const far::indexInfo<rank> &ii, T_x &x){
        T_x_inner &data=far::getItem(x,ii);
        return data*2;
    };
    return far::makeElemental<l>(x);
}

int main(){
    far::Array<double,1> a = far::linspace<double>(1,10,10);
    double d=6.7;
    std::cout << doubler(d) << "\n";
    std::cout << doubler(a) << "\n";
}
```

On its own, this achieves pretty much nothing. You have implemented the same doubler function

but with a more complex syntax. The power of this approach comes from the fact you can access the element inside the `indexInfo` object that has the index in. This is a member called `indices` and is a `std::tuple` of `rank` items, all items being of type `int64_t`. The indices run from 1 to N inclusive in each rank (unless you have changed the default lowerbound, in which case it is always correct to say that it runs from a constant called `defaultLB` to `defaultLB+N-1` inclusive). This means that you could in theory implement a derivative function like this

```
#include <iostream>
#include "far.h"

template<far::anyArray T_x, far::anyTypeArray<1> T_y, std::integral T_dir>
auto deriv(const T_x &x, const T_y&y, T_dir dir){
    constexpr int rank = far::arrayInfo<T_x>::rank;
    auto l = [](const far::indexInfo<rank> &ii, const T_x &x, const T_y &y,
    ↪  const T_dir &dir){
        auto cIndex = far::getTupleLevel(ii.indices,dir);
        if (cIndex == far::defaultLB || cIndex == far::defaultLB +
        ↪  static_cast<int64_t>(x.getRankSize(dir)) - 1) return 0.0;
        //Make shifted version of the indices, one shifted one to the left,
        ↪  and one shifted one to the right
        auto left = ii.indices;
        far::getTupleLevel(left,dir-1)--;
        auto right = ii.indices;
        far::getTupleLevel(right,dir-1)++;
        //This produces offsets in the rank 1 array representing the axis
        std::tuple<int64_t> leftIndex = far::getTupleLevel(left,dir-1),
        ↪  rightIndex = far::getTupleLevel(right,dir-1);
        //Use getItem to actually get the elements
        return (far::getItem(y,right)-
        ↪  far::getItem(y,left))/(far::getItem(x,rightIndex)-
        ↪  far::getItem(x,leftIndex));
    };
    return far::makeElemental<l>(std::forward<const
    ↪  T_x&>(x),std::forward<const T_y&>(y),std::forward<T_dir>(dir));
}

int main(){
    far::Array<double,1> a = far::linspace<double>(0,M_PI*4.0,120), b, c;
    b = far::sin(a);
    c = deriv(a,b,1);
    for (size_t i=2;i<=a.getRankSize(1)-1;i++){
        std::cout << a(i) << " ," << b(i) << " ," << c(i) << std::endl;
```

```
    }
}
```

This example uses a FAR++ helper function `getTupleLevel` which allows you to access a tuple element when the tuple element index is only known at runtime. It also shows that `getItem` can be called with a tuple of `rank` x `int64_t` elements. In this case it creates shifted indices in the specified direction for the `y` (or more accurately independent axis) array and a single element tuple for the rank `1` `x` array. When being used like this the indexer uses the largest rank of a supplied array and the rank sizes of the leftmost array parameter of that rank.

This code is not the nicest to write, but it does give you substantial power to do various things. Code like this is used internally in FAR++ to write routines like `maxloc` that need access to the elements of an array.

#### Non numeric arrays

FAR++ makes as few restrictions as possible on what types an array can be made up of. The main restriction is that the type must be capable of default construction and either copy assignment or copy construction.

**String arrays** You can have an array of strings in three ways : an array of `std::string`, an array of `char[N]` (a fixed length string) and `char*`. All of the arrays can be created in the same way as normal numeric arrays by specifying a type in the template parameters. There are a few special functions that only apply to string arrays. They are overloaded for all of these types

- len - Get the length of an array. Is the length of the string for `std::string`, N for `char[N]` string, and the result of `strlen` for `char*` arrays
- len_trim - Get the length of an array with trailing whitespace trimmed off.
- lle - Returns logical result comparing two strings as being lexically less than or equal to each other
- llt - Returns logical result comparing two strings as being lexically less than each other
- lge - Returns logical result comparing two strings as being lexically greater than or equal to each other
- lgt - Returns logical result comparing two strings as being lexically greater than each other
- repeat - Returns a string consisting of N copies of a source string

**Pointer arrays** Pointer types can be used in general in arrays and work as you might expect. No type conversion is performed, so only pointer arithmetic operations can be performed on the data

in pointer arrays. There are two functions that are designed to help with using pointer arrays. These are

- reference(source_array) - return an array of pointers from an array of values. Each element of the pointer array refers to the matching element of the source array
- dereference(pointer_array) - return a lazy array that returns a reference to the element that the pointer points to. Essentially converts a pointer array to a reference returning lazy array that can be used anywhere that a normal Array can.

**Arrays of types** FAR++ arrays work properly with arbitrary types as array datatypes, so long as the types can be default constructed. FAR++ simply passes through operations to underlying types, so the operators will work as expected so long as your class implements them. As well as allowing you to use your classes element by element with the normal `.` operator, FAR++ provides two features to help you work with arrays of types.

- Component selection - as in Fortran, you can convert an array of types into a lazy array of references to individual members of the type. The syntax isn't as elegant as the Fortran one, but the effect is the same
- Member calling - In Fortran, so long as you make a member function of a type elemental, you can call that member function on all elements of an array of the type in a single call. Once again, the syntax isn't as elegant as in Fortran, but you can do the same here

**Component selection (Alpha)** *This section should be considered alpha again. This feature is going to be maintained because it is in the Fortran standard, but only a small section of all possible use cases has been tested*

A nice feature of Fortran is *component selection*, where an array of types can have its internal members referred to as if they were an array of the member types. For example

```
PROGRAM test

  IMPLICIT NONE

  TYPE :: demo
    INTEGER :: value
  END TYPE

  TYPE(demo), DIMENSION(10) :: elements
  INTEGER :: i
```

```
  elements%value = [(i,i=1,10)]

  PRINT *,elements%value

END PROGRAM
```

In this example, the value members of a type are set and printed as an array. This level of transparency needs compiler support, but a similar general approach has been made in FAR++

```
#include <iostream>
#include "far.h"

using namespace far;

class demo{
    public:
    int ivalue;
    float fvalue;
};

int main(){

    Array<demo,1> a(10);
    for(int i=1;i<=10;i++){
        a(i).ivalue = i;
    }
    std::cout << "Printing integer value using component selection :";
    std::cout <<  a.selectComponent<&demo::ivalue>() << "\n\n";
    std::cout << "Changing floating point component using component
    ↪  selection :";
    a.selectComponent<&demo::fvalue>() = linspace<float>(0.1,1.0,10);
    for(int i=1;i<=far::size(a);i++){
        std::cout << a(i).fvalue << " ";
    }
    std::cout << "\n";

}
```

As you can see FAR++ arrays have a method called `selectComponent` that takes a pointer to a member variable as a template parameter. The result of calling this method is a lazy array returning references to the selected member variables. This and lazy elemental functions returning references are the only times that you can assign values to a lazy array, so you can write a line that assigns values to a selected component as shown above.

So long as the methods are written as elemental, you can also call component methods of a Fortran type. They behave just like any other elemental subprograms, so you can optionally pass arrays to the method as parameters and that is matched up element by element with the array of types, and returns from functions become an array of the results

```
MODULE demoMod

  IMPLICIT NONE
  TYPE :: methodTest
    INTEGER :: i
    CONTAINS
    PROCEDURE set_i
    PROCEDURE get_i
  END TYPE methodTest

  CONTAINS

  ELEMENTAL SUBROUTINE set_i(this,val)
    CLASS(methodTest), INTENT(INOUT) :: this
    INTEGER, INTENT(IN) :: val
    this%i=val
  END SUBROUTINE set_i

  ELEMENTAL FUNCTION get_i(this)
    CLASS(methodTest), INTENT(IN) :: this
    INTEGER :: get_i

    get_i = this%i
  END FUNCTION get_i

END MODULE demoMod

PROGRAM demo

  USE demoMod
  IMPLICIT NONE
  TYPE(methodTest),DIMENSION(10) :: t
  INTEGER :: i
  INTEGER, DIMENSION(10) :: values = [(i,i=1,10)]

  CALL t%set_i(values)
  PRINT *, t%get_i()
```

```
END PROGRAM demo
```

You can also call a member function of a C++ class. If the member function returns a value and doesn't have parameters that can be changed by the call then the return is a lazy array containing the result of the function for each element. If there is no return value or the method parameters can be changed by the call then the method is called immediately for each element. This behaviour is equivalent to that for elemental functions discussed earlier. This function is called callMethod.

```
#include <iostream>
#include "far.h"

class demo{
    private:
    int i;
    public:

    //Setter method
    void set_i(int val){
        i = val;
    }

    int& get_i(){
        return i;
    }

    int inc(int i){
        int temp = ivalue;
        ivalue+=i;
        return temp;
    }

    int inc(){
        int temp = ivalue;
        ivalue++;
        return temp;
    }

};

int main(){

    //Create an array of objects
    far::Array<demo,2> c(3,3);
```

```
    //Use callMethod to set the value of the object from a temporary
    //NB this will not allocate the array like normal assignment!

↪   c.callMethod<&demo::set_i>(far::reshape(far::linspace<int>(1,9,9),3,3));

    //Print the array, again using callMethod
    std::cout << "After setting using callMethod array is \n";
    std::cout << far::gridPrint(c.callMethod<&demo::get_i>()) << "\n\n";

    //Call the version of inc that takes no arguments
    //Note that have to manually cast the method to the correct type
    //to resolve the overload
    std::cout << "Original value of array returned from inc() \n";
    std::cout <<
    ↪   far::gridPrint(c.callMethod<static_cast<int(demo::*)()>(&demo::inc)>())
    ↪   << "\n\n";

    std::cout << "After calling inc() array is \n";
    std::cout << far::gridPrint(c.callMethod<&demo::get_i>()) << "\n\n";


    //Call the version of inc that takes an argument
    std::cout << "Original value of array returned from inc(2) \n";
    std::cout <<
    ↪   far::gridPrint(c.callMethod<static_cast<int(demo::*)(int)>(&demo::inc)>(2))
    ↪   << "\n\n";

    std::cout << "After calling inc(2) array is \n";
    std::cout << far::gridPrint(c.callMethod<&demo::get_i>()) << "\n\n";

}
```

This shows using call method in various ways, including how to deal with overloads for the same function. This, sadly, is not as elegant as in Fortran where you just call the method with the correct parameter and the overload is selected. Here you have to explicitly cast the pointer to the right type to match the function that you want to call.

There is one potential difficulty in C++ that Fortran doesn't have. In C++ it is possible for a function to return a reference to a value rather than a value. The `getIValue` method listed here has no changing parameters and returns a reference, so it becomes a lazy array. Returning references from a lazy array is simple - when each element is requested, a function is called to evaluate the result and that function returns the reference, but what happens if someone writes a method that changes it's

parameters but returns references? Well, you can force `callMethod` to return a lazy array anyway using `far::ert::Lazy` as a second template parameter just as you can with `makeElemental`, but this has all of the problems described in the elemental function section. What you want to have is an elemental function that returns an array of references, but that isn't possible directly because references cannot be default constructed.

To work around this problem, we introduce the general idea of an array of *late binding references*.

**Reference arrays (Alpha feature)** To allow you to have arrays of references, FAR++ introduces the idea of a *late binding reference*. This isn't quite the same idea as late binding in general [https://en.wikipedia.org/wiki/Late_binding] but represents a vaguely similar idea, in that the true value to which the reference refers is determined when it is first assigned to, not when the reference is created. So to clarify

- Before you assign a value to a late binding reference it is invalid to do anything else with it and you will trigger a `std::runtime_error` exception if you try to
- When you first assign to a late binding reference it captures a reference to the value being assigned
- You can now access that reference or assign to it as if it was a normal reference

This is implemented using a simple class called `LBrefWrapper`, but this class should never turn up in your code, so we will gloss over it's properties. To create a reference array, you just create an array of references. Behind the scenes, this is automatically converted to use the LBrefWrapper and automatically converted back before it enters your code again.

```
#include <iostream>
#include "far.h"

int main(){

    //Create a normal array
    far::Array<int ,2> A = far::reshape(far::linspace(1,10,10),2,5);
    //Create an array of references
    far::Array<int&,2> B;

    //This is the key line. This binds the array of references to the array
    ↪  A
    //After this is done, B is a reference to A and cannot be made to refer
    ↪  to anything else
    //Any changes to B now propagate to A
    B=A;

    std::cout << far::gridPrint(B) << "\n";
```

```
    //This changes the value of A
    B=14;

    std::cout << far::gridPrint(A) << "\n";

}
```

This example shows how reference arrays work. By simply assigning another array to a reference array the references are "locked" to the array that was assigned to it. After that, the reference array can be used anywhere that a normal array of the type can be used, just as a reference can be used anywhere that a reference is. All of the packing and unpacking of data from the `LBrefWrapper` class is done automatically.

Arrays of references are a rather odd idea that breaks many of the properties that arrays are supposed to have. Performance would be much better if you had a FAR++ pointer to A (`B.pointTo(A)`) or even a C++ reference (or even pointer) to A rather than an array of references to each element since you would have only one extra indirection as well as potentially much better memory locality. This shows how to create and use an array of references but generally it is expected that these will only be used to handle the results of elemental functions and called methods that return references.

#### Selected kinds

While nowadays in Fortran it is common for variable sizes to be hard specified using kind specifiers like `REAL32` or `INT64` there is a slightly older mechanism that is still useful. This is the mechanism of selected kinds. There are two

- `selected_int_kind(R)` - Get a kind that can store values at least between 10^-R and 10^R
- `selected_real_kind(P,R,RADIX)` - Get a kind that has a decimal precision of at least P digits, an exponent range of at least R and the specified radix. P, R and RADIX are all optional with P and R are assumed 0 if missing and radix is assumed to be the default radix if it is not specified.

The returned kinds can be used to specify the kinds of integers, reals and complex numbers

The equivalents in FAR++ are templates, and have the following form

- `selected_int_kind<R>` - Get a type that can store integer values at least between 10^-R and 10^R
- `selected_uint_kind<R>` - Equivalent to `std::create_unsigned_t<selected_int_kind<R>>`
- `selected_real_kind<P,R>` - Get a type that can store floating point values with a decimal precision of at least P digits and an exponent range of at least R with the default radix
- `selected_real_kind_p<P>` - Equivalent to `selected_real_kind<P,0>`

- `selected_reak_kind_r<R>` - Equivalent to `selected_real_kind<0,R>`
- `selected_complex_kind<P,R>` - Get a type that can store complex values with the real and imaginary parts being of a type selected using `selected_real_kind<P,R>`
- `selected_complex_kind_p<P>` - Equivalent to `selected_complex_kind<P,0>`
- `selected_complex_kind_r<R>` - Equivalent to `selected_complex_kind<0,R>`

```
#include <iostream>
#include "far.h"

int main(){

    {
        using kind = far::selected_int_kind<1>;
        std::cout << "Size of selected_int_kind<1> is " << sizeof(kind) <<
        ↪  std::endl;
    }

    {
        using kind = far::selected_int_kind<2>;
        std::cout << "Size of selected_int_kind<2> is " << sizeof(kind) <<
        ↪  std::endl;
    }

    {
        using kind = far::selected_int_kind<4>;
        std::cout << "Size of selected_int_kind<4> is " << sizeof(kind) <<
        ↪  std::endl;
    }

    {
        using kind = far::selected_int_kind<8>;
        std::cout << "Size of selected_int_kind<8> is " << sizeof(kind) <<
        ↪  std::endl;
    }

    {
        using kind = far::selected_int_kind<16>;
        std::cout << "Size of selected_int_kind<16> is " << sizeof(kind) <<
        ↪  std::endl;
    }

#ifdef FAR_INT128
    {
```

```
        using kind = far::selected_int_kind<32>;
        std::cout << "Size of selected_int_kind<32> is " << sizeof(kind) <<
        ↪  std::endl;
    }
#endif

}
```

by default, the `int_fast*_t` types are used by `selected_int_kind`, so you may see only a small variety of sizes from this code. If you want to use the default `int*_t` types then define the preprocessor variable `FAR_EXACT_INT_KIND`. If FAR++ can detect support for larger integers than `int64_t` then it will set the `FAR_INT128` preprocessor variable to that type. If you know that your compiler supports larger integer variables than `int64_t` then you can manually define `FAR_INT128` to that type to enable support

### Parallelism

**Threads** The design philosophy with FAR++ and threaded parallelism is to make it so that FAR++ adds as few restrictions as possible. There is no global state to the arrays in FAR++, so arrays created in different threads are fully thread safe. Allocating and deallocating arrays that are shared between threads is not thread safe, and multiple threads seeking to do so at the same time can cause crashes and/or memory leaks. Reading elements of shared arrays, whether by index, whole array operation or slicing, are thread safe, so long as no stores to the array occur. Writing to shared arrays should only cause crashes or memory errors if using whole array assignment that causes a reallocation of the shared array, but it is up to the user to ensure correctness of the results. Otherwise, the normal rules for multithreaded access to variables apply.

This means that FAR++ arrays can be used with any CPU thread based parallelism system, such as C++ threads, Intel TBB or others. Unfortunately, there is no threaded acceleration built in to any array operators or functions, so you will have to write your own threaded code using the array indices.

**OpenMP (Alpha feature)** One of the really nice elements of Fortran with OpenMP is the `WORKSHARE` OpenMP directive. With `WORKSHARE` you can use a Fortran array operation and (if possible) it will simply automatically parallelise without you having to write any code.

In FAR++ we have tried to replicate this behaviour, but without explicit compiler support this involves some slightly … interesting approaches to implementing it. In userspace it should be seamless, and it supports all of the elements of OpenMP that we can think of but if you use unusual enough feature of OpenMP this might break.

```
#include <iostream>
#include "far.h"
#include <omp.h>

int get_thread_num(){
  #ifdef _OPENMP
  return omp_get_thread_num();
  #else
  return 0;
  #endif
}

int main(){

    int threads = 1;
    #ifdef _OPENMP
    threads=omp_get_max_threads();
    #endif
    far::Array<int,1> a(threads*3);
    #pragma omp parallel
    {
        far::beginWorkshare();
        a=get_thread_num();
        far::endWorkshare();
    }

    std::cout << "a = " << a << std::endl;
}
```

This is a classical OpenMP example although usually written with a loop rather than using workshare. You simply populate each element of the array with the OpenMP thread ID for the thread that is working on that element.

You can write the same code in Fortran, although since `omp_get_thread_num()` is not an elemental function you have to write an interposer function to allow it to work.

```
PROGRAM omp_num

  USE omp_lib
  INTEGER, DIMENSION(:), ALLOCATABLE :: a
  INTEGER :: threads = 1
  !$ threads=omp_get_max_threads()

  ALLOCATE(a(threads*3))
```

```
  !$OMP PARALLEL
  !$OMP WORKSHARE
    a=get_thread_num()
  !$OMP END WORKSHARE
  !$OMP END PARALLEL

  PRINT '(A,*(I0,1X))',"a = ", a

  CONTAINS

  IMPURE ELEMENTAL FUNCTION get_thread_num()
    INTEGER :: get_thread_num
    get_thread_num = 0
    !$ get_thread_num = omp_get_thread_num()
  END FUNCTION get_thread_num

END PROGRAM omp_num
```

Because of how `workshare` works, it is almost a necessity that scheduling is static, with the same array elements always being operated on by the same thread.

In general, workshare can be used in much the same way that it is used in Fortran. As a final note, it must be stressed that this is very definitely an alpha part of FAR++ and while it has worked for all current tests that have been performed using it, we recommend that it is used only for very simple things like assigning one array to another where we have extensively tested it. Any use of the workshare directive should be very carefully checked for correctness.

FAR++ workshare should work correctly when used in different teams, but testing for this has been minimal.

**MPI** There is currently no direct support for MPI in FAR++, although future extensions will likely add MPI features. Currently the easiest way to use MPI is to make use of the `makeCopyIn`, `makeCopyOut` and `makeCopyInOut` functions.

```
#include <iostream>
#include "far.h"
#include <mpi.h>
#include <chrono>
#include <thread>

int main(int argc, char **argv){

    MPI_Init(&argc, &argv);
```

```
    int nproc;
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    if (nproc !=2){
        std::cerr << "This demo only works on two ranks\n";
        MPI_Abort(MPI_COMM_WORLD,-1);
    }
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        if (rank==0) {
            std::cout << "This shows swapping edge cells as is often used
            ↪  in finite difference schemes. To prevent the prints from
            ↪  overlapping the second rank waits for 100ms before
            ↪  printing\n";
        }

    far::Array<int,2> a(far::Range(0,4),far::Range(0,4));
    if (rank==0){
        a=reshape(far::linspace(1, 25, 25),5,5);
    } else {
                a=-reshape(far::linspace(1,25,25),5,5);
    }

        //Copy left hand edge to right guard cells
    MPI_Sendrecv(far::makeCopyIn(a(far::Range(1,3),1)),3,MPI_INT,1-
↪   rank,0,far::makeCopyOut(a(far::Range(1,3),4)),3,MPI_INT,1-
↪   rank,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        //Copy right hand edge to left guard cells
        MPI_Sendrecv(far::makeCopyIn(a(far::Range(1,3),3)),3,MPI_INT,1-
↪   rank,0,far::makeCopyOut(a(far::Range(1,3),0)),3,MPI_INT,1-
↪   rank,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

        std::this_thread::sleep_for(std::chrono::milliseconds(rank*100));
    std::cout << far::gridPrint(a) << "\n\n";

    MPI_Finalize();

}
```

This is comparable to (but less elegant than) using array slice in Fortran to MPI calls, but has the same major problem - you cannot use it in any of the asynchronous or RDMA routines because the lifetime of the copy cannot is no longer sufficient. As described in the section on `makeCopy*` you can do this manually by storing the result of the `makeCopy*` to a variable and then using the `copyBack`

method to copy the result back to the true array when needed. This is generally not a good idea, so we recommend using MPI_Types, much as you would in Fortran, combined with judicious use of the `data` method of the array. Remember that this will *only* work if the array is contiguous. If it is non-contiguous then you will not be able to call the `data` method. If you reach the point of needing to use an `MPI_Type` to send a subsection of an already sliced array, then this would be difficult in any language. Eventually, we are planning to add a feature to allow you to directly get an MPI type that would correspond to any slice that you have performed on the array, but this feature is not in imminent danger of release

**Coarrays** Fortran has a built in parallel programming model called *coarray Fortran*. This is a partitioned global address space model where you can access elements of an array on other processors by using one or more additional indices specified in square brackets. At present FAR++ has no support at all for Coarrays, but the square bracket operator of the `Array` and `lazyArray` classes has been reserved against possible future implementation of this feature. Because of the stringent requirements for performance for this feature to be useful, it is likely that substantial further development will be needed before coarrays are implemented in FAR++.

**GPUs** Currently FAR++ will not work with any accelerator library that requires flagging functions with prefixes to create device code. Making FAR++ work on an accelerator is something that we want to cover in future.

#### Array bounds checking

Most Fortran compilers have support for array bounds checking - checking if an array is accessed outside of it's bounds. A version of this behaviour is also supported in FAR++. The equivalent of Fortran's behaviour is obtained by setting the precompiler flag `FAR_DEFAULT_BOUNDS_CHECK`

```
#include <iostream>
#define FAR_DEFAULT_BOUNDS_CHECK
#include "far.h"

int main() {

    far::Array<int,1> a(10);
    try{
        a(0) = 1;
    } catch (std::out_of_range& e) {
        std::cout << "Caught exception: " << e.what() << std::endl;
    }
```

```
}
```

Accessing an out of range element, either by directly accessing it like this or by applying an array function to arrays of different sizes will cause an `std::out_of_range` error to be thrown. The error message will say what element in what index is out of range, but it cannot say which array has been accessed out of range.

You can also set an individual array to have different array bounds checking to the default behaviour. This is done by setting the third template parameter to one of three values

- `far::bounds_check_state::bc_always` - always array bounds check this array
- `far::bounds_check_state::bc_never` - never array bounds check this array
- `far::bounds_check_state::bc_default` - explicitly use default array bounds checking

```
#include <iostream>
#include "far.h"

int main() {

    far::Array<int,2> a(10,10);
    far::Array<int,2,far::bounds_check_state::bc_always> b(10,10);

    a=1;b=2;

    //This will not cause any problems because it will calculate a point
    //Within the bounds of the underlying storage, but it is still
     ↪  incorrect
    //No error will be thrown because bounds checking is off for this array
    try{
        std::cout << "Out of range element in unchecked array is " <<
         ↪  a(11,1) << std::endl;
    } catch (std::out_of_range& e) {
        std::cout << "Caught exception in unchecked array: " << e.what() <<
         ↪  std::endl;
    }

    //This will cause an error because bounds checking is on for this array
    try{
        std::cout << "Out of range element in checked array is " << b(11,1)
         ↪  << std::endl;
    } catch (std::out_of_range& e) {
```

```
        std::cout << "Caught exception in checked array: " << e.what() <<
        ↪  std::endl;
    }

    a=b;
    std::cout << "Unchecked array has been assigned a value from checked
    ↪  array\n";
    std::cout << "Smallest element in unchecked array is " <<
    ↪  far::minval(a) <<"\n";
    std::cout << "Smallest element in checked array is " << far::minval(b)
    ↪  << "\n";

}
```

#### Fort files

This is last because it isn't technically in any Fortran standard, even though it is widely supported by extant compilers and is permitted in the Fortran standard. It is permitted for logical unit numbers (LUNs) to be *preconnected* to specific inputs or outputs even if they can be manually connected to a different inputs or outputs. Some of these are preconnected to things like stdin, stderr and stdio, but for many compilers all non-special LUNs are preconnected to a file called `fort.{lun}` where `{lun}` is replaced with the LUN specified. This means that reading or writing to LUN 10 without expicitly opening it will read or write to the file `fort.10`. This is very helpful for debugging, so we have replicated this in FAR++. To use it, simply call `fortFile(lun)` which returns you a reference to an fstream object which is already opened on a file called `fort.{lun}`. You can freely close the file as required and it will be reopened if needed on the next call to `fortFile`

```
#include <iostream>
#include "far.h"

int main(){

    auto &output = far::fortFile(20);
    output << "Hello world";
    output.close();

}
```

### Case Study

As a simple case study for using FAR++ in anger on a real(ish) problem, we will consider solving the heat equation by Jacobi iteration. There is no point in reproducing the derivation of this method. In 2D it essentially boils down to having two arrays, a main array and a working array, setting each element of the working array to the average of the four cells in the main array with neighbouring indices in the two coordinate axes and then copying that working array back to the main array after all cells have been updated. There is a single strip of guard cells around the domain that are used as boundary conditions. This is then iterated repeatedly until a desired tolerance is reached. In this simple case study, this convergence test is replaced with a simple fixed number of iterations since the calculation of the tolerance is not interesting.

Taking a simple Fortran implementation of this problem we have

```
PROGRAM jacobi
  USE ISO_FORTRAN_ENV
  IMPLICIT NONE
  INTEGER, PARAMETER :: NX=100, NY=100, NITS=2000
  REAL(REAL64), DIMENSION(:,:), ALLOCATABLE,TARGET :: data, data2
  INTEGER :: i, j, its, start,end
  REAL :: count_rate
  ALLOCATE(data(0:NX+1,0:NY+1), data2(0:NX+1,0:NY+1))
  data=0
  data(0,:)=1
  data(NX+1,:)=1
  data(:,0)=1
  data(:,NY+1)=1
  CALL SYSTEM_CLOCK(start,count_rate)
  DO its=1,NITS
    DO j=1,NY
      DO i=1,NX
        data2(i,j)=0.25_REAL64 *
↪   (data(i-1,j)+data(i+1,j)+data(i,j-1)+data(i,j+1))
      ENDDO
    ENDDO
    data(1:NX,1:NY)=data2(1:NX,1:NY)
  ENDDO
  CALL SYSTEM_CLOCK(end)
  PRINT *, 'Time for main loop is ', REAL(end-start)/count_rate
  PRINT '(F24.17)', MAXVAL(data(1:NX,1:NY))
  PRINT '(F24.17)', MINVAL(data(1:NX,1:NY))
  PRINT '(F24.17)',
  ↪   SUM(data(1:NX,1:NY))/REAL(SIZE(data(1:NX,1:NY),KIND=INT64),REAL64)
```

```
END PROGRAM jacobi
```

It is easy to see both algorithmic (SOR Gauss-Seidel) and programmatic (use pointers to switch the active array rather than copying back) improvements but none of these are particularly illustrative of how to compare Fortran to FAR++.

Let us compare this with a comparable implementation in C++ using FAR++

```
#include <iomanip>
#include "far.h"
#include "timer.h"
#define NX 100
#define NY 100
#define ITS 2000

using namespace far;

int main(){
    Array<double,2> data(Range(0,NX+1),Range(0,NY+1)), data2;
    data=0;
    data2.mold(data);
    //Set the boundaries on data
    data(0,Range(0,NY+1))=1;
    data(NX+1,Range(0,NY+1))=1;
    data(Range(0,NX+1),0)=1;
    data(Range(0,NX+1),NY+1)=1;

    timer t;
    t.begin("main loop");
        for (int its = 0; its<ITS;++its){
            for (int j=1;j<=NY;++j){
                for (int i=1;i<=NX;++i){
                    data2(i,j) = (data(i-1,j) + data(i+1,j) + data(i,j-1) +
↪   data(i,j+1)) * 0.25;
                }
            }
            //Copy back
            data(Range(1,NX),Range(1,NY))=data2(Range(1,NX),Range(1,NY));
        }
    t.end();
    std::cout << std::setprecision(17);
    std::cout << maxval(data(Range(1,NX),Range(1,NY))) << "\n";
    std::cout << minval(data(Range(1,NX),Range(1,NY))) << "\n";
```

```
    std::cout <<
    ↪  sum(data(Range(1,NX),Range(1,NY)))/float(data(Range(1,NX),Range(1,NY)).getSize()
    ↪  << "\n";
}
```

It is immediately clear that the FAR++ implementation of this algorithm is very similar to the Fortran version. Printing the outputs from the two codes and putting them side by side we find that the results are identical, since on the platform this code was run onx the C++ and Fortran variable types are exactly the same.

```
Fortran                 C++
0.99937931483774856     0.99937931483774856
0.39276316387216581     0.39276316387216581
0.74423981043941689     0.74423981043941689
```

This can be compared with an implementation using C++ native heap arrays.

```
#include <string>
#include <iomanip>
#include <iostream>
#include "timer.h"

#define NX 100
#define NY 100
#define ITS 2000

#define index(i,j) i*(NY+2)+j

int main(){
    double *data = new double[(NX+2)*(NY+2)];
    double *data2 = new double[(NX+2)*(NY+2)];
    for (int i = 0; i<(NX+2)*(NY+2);++i){
        data[i]=0;
        data2[i]=0;
    }
    //Set the boundaries on data
    for (int i = 0; i<NX+2;++i){
        data[index(i,0)]=1;
        data[index(i,NY+1)]=1;
    }
    for (int j = 0; j<NY+2;++j){
        data[index(0,j)]=1;
```

```
        data[index(NX+1,j)]=1;
    }
    timer t;
    t.begin("main loop");
    for (int its = 0; its<ITS;++its){
        for (int i = 1; i<NX+1;++i){
            for (int j = 1; j<NY+1;++j){
                data2[index(i,j)] = (
                        data[index(i-1,j)] +
                        data[index(i+1,j)] +
                        data[index(i,j-1)] +
                        data[index(i,j+1)]) * 0.25;
            }
        }
        for (int i = 1; i<NX+1;++i){
            for (int j = 1; j<NY+1;++j){
                data[index(i,j)]=data2[index(i,j)];
            }
        }
    }
    t.end();
    std::cout << std::setprecision(17);

    double mx = std::numeric_limits<double>::min();
    double mn = std::numeric_limits<double>::max();
    double sum = 0;
    for (int i = 1; i<NX+1;++i){
        for (int j = 1; j<NY+1;++j){
            mx = std::max(mx,data[index(i,j)]);
            mn = std::min(mn,data[index(i,j)]);
            sum += data[index(i,j)];
        }
    }
    sum /= double(NX*NY);
    std::cout << mx << std::endl;
    std::cout << mn << std::endl;
    std::cout << sum << std::endl;

}
```

While this version is poor C++ in many ways including because it uses new to allocate memory etc. improving these problems will not improve the size of the code or reduce the boilerplate. The result of this code is again identical to the Fortran and FAR++ versions.

So FAR++ and Fortran allow you to write code that looks very similar in both languages, but in Fortran one has a different option to write this - express the shifted arrays in terms of array subsections.

```
PROGRAM test
  USE ISO_FORTRAN_ENV
  IMPLICIT NONE
  INTEGER, PARAMETER :: NX=500, NY=500, NITS=20000
  REAL(REAL64), DIMENSION(:,:), ALLOCATABLE,TARGET :: T, T2
  INTEGER :: i, j, its, start,end
  REAL :: count_rate
  ALLOCATE(T(0:NX+1,0:NY+1), T2(0:NX+1,0:NY+1))
  T=0
  T(0,:)=1
  T(NX+1,:)=1
  T(:,0)=1
  T(:,NY+1)=1
  CALL SYSTEM_CLOCK(start,count_rate)
  DO its=1,NITS
    T2(1:NX,1:NY)=0.25_REAL64 *
↪   (T(0:NX-1,1:NY)+T(2:NX+1,1:NY)+T(1:NX,0:NY-1)+T(1:NX,2:NY+1))
    T(1:NX,1:NY)=T2(1:NX,1:NY)
  ENDDO
  CALL SYSTEM_CLOCK(end)
  PRINT *, 'Time for main loop is ', REAL(end-start)/count_rate
  PRINT '(A,F24.17)', "Maxval :", MAXVAL(T(1:NX,1:NY))
  PRINT '(A,F24.17)', "Minval :", MINVAL(T(1:NX,1:NY))
  PRINT '(A,F24.17)', "Mean   :",
  ↪  SUM(T(1:NX,1:NY))/REAL(SIZE(T,KIND=INT64))

END PROGRAM test
```

This syntax is rather more compact than the loop based syntax, and it can also be implemented in C++ using FAR++

```
#include <iomanip>
#include "far.h"
#include "timer.h"
#define NX 500
#define NY 500
#define ITS 20000

using namespace far;

int main(){
```

```
    Array<double,2> data(Range(0,NX+1),Range(0,NY+1)), data2;
    data=0;
    data2.mold(data);
    //Set the boundaries on data
    data(0,Range(0,NY+1))=1;
    data(NX+1,Range(0,NY+1))=1;
    data(Range(0,NX+1),0)=1;
    data(Range(0,NX+1),NY+1)=1;

    timer t;
    t.begin("main loop");
  for (int its = 0; its<ITS;++its){
    //Update the temporary
    data2(Range(1,NX),Range(1,NY)) = (
        data(Range(0,NX-1),Range(1,NY)) +
        data(Range(2,NX+1),Range(1,NY)) +
        data(Range(1,NX),Range(0,NY-1)) +
        data(Range(1,NX),Range(2,NY+1))) * 0.25;
    //Copy back
    data(Range(1,NX),Range(1,NY))=data2(Range(1,NX),Range(1,NY));
  }
    t.end();
    std::cout << std::setprecision(17);
    std::cout << "Maxval :" << maxval(data(Range(1,NX),Range(1,NY))) <<
    ↪  "\n";
    std::cout << "Minval :" << minval(data(Range(1,NX),Range(1,NY))) <<
    ↪  "\n";
    std::cout << "Mean   :" <<
    ↪  sum(data(Range(1,NX),Range(1,NY)))/float(data.getSize()) << "\n";
}
```

The slightly longer syntax for selecting array slices mean that the FAR++ code isn't quite as compact, but it is still quite readable and involves much less boilerplate code than using the array indices. Once again, both codes produce identical results.

Both approaches can be generalised to higher dimensions easily.

### Performance

FAR++ is not primarily intended as a tool for large scale simulation, so absolute maximum performance is not the primary aim, but performance must be broadly comparable to Fortran and native C++. A variety of test cases have been used to validate the performance of FAR++, but the simplest is just to

take the above case studies, increase the number of iterations and time the output. FAR++ uses fairly modern template features of C++, so newer compilers tend to work better. These test results were done with g++ 14.1.0 and gfortran 14.1.0 on an AMD Epyc 7443P with 128GB of RAM. Very similar relative results were found with GCC 12 and 13 series compilers. GCC 11 is able to compile the code but the time taken to complete the C++ tests increases by about 10%. Each run is done 5 times and the time is averaged. For both compilers these optimisation flags were used `-O3 -flto -march=native`. All ± figures are at 1 standard deviation

| NX=NY=N | N_Iterations | C++ Native | Fortran Indices | FAR++ Indices | Fortran Slices | FAR++ Slices |
|---|---|---|---|---|---|---|
| 100 | 2000000 | 7.69±0.20 | 7.61±0.03 | 7.56±0.041 | 7.58±0.01 | 9.05±0.15 |
| 500 | 80000 | 9.90±0.60 | 9.80±0.35 | 9.94±0.31 | 10.31±0.51 | 10.09±0.40 |
| 1000 | 20000 | 9.87±0.46 | 9.88±0.49 | 9.94±0.49 | 9.94±0.44 | 9.80±0.49 |
| 10000 | 200 | 19.39±0.09 | 19.40±0.21 | 19.43±0.05 | 19.33±0.22 | 18.53±0.15 |

As you can clearly see, for this simple but illustrative test case, FAR++ is able to perform comparably to both Fortran and native C++ arrays when using array indices for all problem sizes. For small problem sizes FAR++ array slice notiation is about 20% slower than Fortran array slice notation, but this seems to be due to a fixed overhead for using array slices and disappers at even 500x500 problems.

Moving to 3D sees a fairly variable picture, probably mostly because of the low algorithmic intensity of the test problem, but it is clear that once you get past quite small arrays the performance of FAR++ is comparable to other options.

| NX=NY=NZ=N | N_Iterations | C++ Native | Fortran Indices | FAR++ Indices | Fortran Slices | FAR++ Slices |
|---|---|---|---|---|---|---|
| 10 | 20000000 | 8.33±0.07 | 9.58±0.10 | 9.19±0.05 | 25.91±0.01 | 31.01±0.58 |
| 100 | 20000 | 11.05±0.61 | 11.39±0.47 | 11.20±0.42 | 10.99±0.09 | 12.52±0.63 |
| 500 | 160 | 27.23±0.39 | 26.72±0.58 | 26.91±0.17 | 26.47±0.18 | 20.38±0.03 |
| 1000 | 20 | 26.35±0.52 | 31.30±0.29 | 30.94±0.20 | 30.84±0.14 | 27.08±0.20 |

### OpenMP Performance (Results from alpha feature)

Mostly parallel performance is related to the specific implementation of the parallel algorithm but since the above test case can be parallelised by using the `workshare` directive we can test how that

performs in Fortran and FAR++. Because of the low algorithmic intensity of this problem we will test with the 10000x10000 2D test. Modifying the code to use workshare is very simple in both Fortran and FAR++.

```
#include <iomanip>
#include "far.h"
#include "timer.h"
#define NX 10000
#define NY 10000
#define ITS 200

using namespace far;

int main(){
    Array<double,2> data(Range(0,NX+1),Range(0,NY+1)), data2;
    data=0;
    data2.mold(data);
    //Set the boundaries on data
    data(0,Range(0,NY+1))=1;
    data(NX+1,Range(0,NY+1))=1;
    data(Range(0,NX+1),0)=1;
    data(Range(0,NX+1),NY+1)=1;

    timer t;
    t.begin("main loop");
#pragma omp parallel
    {
        for (int its = 0; its<ITS;++its){
            beginWorkshare();
            //Update the temporary
            (data2(Range(1,NX),Range(1,NY))) = (
                    data(Range(0,NX-1),Range(1,NY)) +
                    data(Range(2,NX+1),Range(1,NY)) +
                    data(Range(1,NX),Range(0,NY-1)) +
                    data(Range(1,NX),Range(2,NY+1))) * 0.25;
            //Copy back
            data(Range(1,NX),Range(1,NY))=data2(Range(1,NX),Range(1,NY));
            endWorkshare();
            #pragma omp barrier
        }
    }
    t.end();
    std::cout << std::setprecision(17);
```

```
    std::cout << "Maxval :" << maxval(data(Range(1,NX),Range(1,NY))) <<
    ↪  "\n";
    std::cout << "Minval :" << minval(data(Range(1,NX),Range(1,NY))) <<
    ↪  "\n";
    std::cout << "Mean   :" <<
    ↪  sum(data(Range(1,NX),Range(1,NY)))/float(data(Range(1,NX),Range(1,NY)).getSize()
    ↪  << "\n";
}
```

| N CPUs | Fortran | FAR++ |
|---|---|---|
| 1 | 18.88±0.18 | 18.91±0.13 |
| 2 | 11.01±0.16 | 11.02±0.06 |
| 4 | 9.99±0.18 | 9.88±0.36 |
| 8 | 8.14±0.47 | 8.17±0.30 |
| 16 | 8.57±0.37 | 8.31±0.37 |

As you can clearly see scaling is quite poor, but is very similar between Fortran and FAR++.

### Conclusion

The aim of FAR++ was to create a library that allowed you to make use of arrays in much the same way as Fortran does with good enough performance to make it useful at least in smaller scale simulations. We believe that FAR++ acheives these aims.

If you find a bug in FAR++ please report it on the Github page. In order of severity bugs can be classed as

1) Incorrect results from FAR++
2) Crashes
3) Inconsistency with the Fortran standard
4) Missing elements of the Fortran standard as it applies to arrays

In the first three cases, please be sure to provide minimal code that demonstrates the problem and a description of what the expected behaviour is. In the fourth case, please specify what in the Fortran standard is missing

If you have additional features that you would like to see in FAR++, please also file them as a feature request on Github