rarray: Multidimensional Runtime Arrays for C++

Ramses van Zon

February 2, 2017 (version 1.2)

1 For the impatient: the what, why and how of rarray

What:

Rarray provides multidimensional arrays with dimensions determined at runtime.

What not:

No linear algebra, overloaded operators etc.

Why:

Usually faster than alternatives,

Uses the same accessors as compile-time (automatic) arrays,

Data is guarranteed to be contiguous for easy interfacing with libraries (CBLAS, LAPACKE).

How:

The header file `rarray` provides the type `rarray<T,R>`, where `T` is any type and `R` is the rank. Element access uses repeated square brackets. Copying rarrays or passing them to functions mean shallow copies, unless explicitly asking for a deep copy. For io, use the additional header file `rarrayio`.

Define a `n`×`m`×`k` array of `floats`:	`rarray<float,3> b(n,m,k);`
Define a `n`×`m`×`k` array of `floats` with data pre-allocated at a pointer `ptr`:	`rarray<float,3> c(ptr,n,m,k);`
Element i,j,k of the array `b`:	`b[i][j][k]`
Pointer to the contiguous data in `b`:	`b.data()`
Extent in the `i`th dimension in `b`:	`b.extent(i)`
Shallow, reference-counted copy of the array:	`rarray<float,3> d=b;`
Deep copy of the array:	`rarray<float,3> e=b.copy();`
A rarray using an existing automatic array:	`float f[10][20][8]={...};` `rarray<float,3> g=RARRAY(f);`
A rarray copy of an existing automatic array:	`rarray<float,3> h=RARRAY(f).copy();`
Output a rarray to screen:	`std::cout << h << endl;`
Read a rarray from keyboard:	`std::cin >> h;`

Contents

2 Introduction

While C and thus C++ has some support for multidimensional arrays whose sizes are known at compile time, the support for arrays with sizes that are known only at runtime, is limited. For one-dimensional arrays, C++ has a reasonable allocation construction in the operators `new` and `delete`. A standard way to allocate a one-dimensional array is as follows:

```
float* a;
int n = 1000;
a = new float[n];
a[40] = 2.4;
delete[] a;
```

It is important to note that this code also works if `n` was not known yet, e.g., if it was passed as a function argument or read in as input.

In the above code snippet, the new/delete construct assigns the address of the array to a pointer. This pointer does not remember its size, so this is not really an 'array'. The standard C++ library does provide a one-dimensional array that remembers it size in the form of the `std::vector`, e.g.

```
const int n = 1000;
std::vector a(n);
a[40] = 2.4;
a.clear();
```

Multi-dimensional runtime-allocated arrays are not supported by C++. The textbook C++ solution for multidimensional arrays that are dynamically allocated during runtime, is as follows:

```
float*** a;
a = new float**[dim0];
for (int i=0;i<dim0;i++) {
  a[i] = new float*[dim1];
  for (int j=0;j<dim1;j++)
    a[i][j] = new float[dim2];
}
```

Drawbacks of this solution are the non-contiguous buffer for the elements (so it's unusable for many libraries) and having to keep track of array dimensions. At first, there seems to be no shortage of libraries to fill this lack of C++ support for dynamic multi-dimensional arrays, such as

- Blitz++;
- The Boost Multidimensional Array Library (`boost::multiarray`);
- Eigen;
- Armadillo; and
- Nested `vectors` from the Standard Template Library.

These typically do have some runtime overhead compared to the above textbook solution, or do not allow arbitrary ranks. In contrast, the purpose of the rarray library is to be a minimal interface for runtime multidimensional arrays with *minimal to no performance overhead* compared to the textbook solution.

Example:

```
#include <rarray>
int main()
{
   rarray<float,3> a(256, 256, 256);
   a[1][2][3] = 105;
}
```

Design Points

1. To have dynamically allocated multidimensional arrays that combine the convenience of automatic c++ arrays with that of the typical textbook dynamically allocated pointer-to-pointer structure.

   The compatibility requirement with pointer-to-pointer structures is achieved by allocating a pointer-to-pointer structure. This accounts for most of the memory overhead of using rarray.
2. To be as fast as pointer-to-pointer structures.
3. To have rarrays know their sizes, so that can be passed to functions as a single argument.
4. To enable interplay with libraries such as BLAS and LAPACK: this is achieved by guarranteeing contiguous elements in the multi-dimensional array, and a way to get this data out.

   Relatedly, it should be allowed to use an existing buffer.

   The guarrantee of contiguity means strided arrays are not supported.
5. To optionally allow bounds checking.
6. To avoid some of the cluttered sematics around `const` correctness when converting to pointer-to-pointer structers.

Features of rarray:

- Can use any data type `T` and any rank `R`.
- Elements are accessible using repeated square brackets, like C/C++ arrays.
- Views on pre-allocated contiguous arrays.
- Does shallow, reference counted, copies by default, but also has a deep `copy` method.
- Use of move semantics when used in c++11 (for efficiency).
- Can be emptied with the `clear` method.
- Can be filled with a uniform value with the `fill` method.
- Can be reshaped.
- Automatic C-style arrays can be converted to rarrays using `RARRAY`.
- Checks index bounds if the preprocessor constant `RA_BOUNDSCHECK` is defined.

- A method `is_clear` to check if the rarray is uninitialized.
- A method to get the number of elements in each dimension (`extent`), or in all dimensions (`shape`).
- A method to obtain the total number of elements in the array (`size`).
- A method to make the data type of the array const (`const_ref`).
- Conversion methods using the member function `data()` for conversions to a `T*` or `const T*`, using the method `ptr_array()` for conversions to `T*const*` or `const T*const*`, and using the method `noconst_ptr_array()` for the conversion to a `T**`.
- Streaming input and output throught the header file `rarrayio`.

3 Using rarrays

3.1 Defining a multidimensional rarray

To use `rarray`, first include the header file:

```
#include <rarray>
```

This defines the (template) classes `rarray<T,R>`, where `T` is the element type, and `R` is the rank (a positive integer). Instances can now be declared as follows:

```
rarray<float,3> s(256,256,256);
s[1][2][3] = 105;
// do whatever you need with s
```

or, using an external, pre-allocated buffer, as

```
float* pre_alloc_data=new float[256*256*256];
rarray<float,3> s(pre_alloc_data,256,256,256);
s[1][2][3] = 105;
// do whatever you need with s
delete[] pre_alloc_data;
s.clear();
```

Without the `delete[]` statement in the latter example, there would be a memory leak. This reflects that rarray is in this case not responsible for the content. The data pointer can also be retrieved using `s.data()`. The `s.clear()` statement ensures there are no dangling references to this data left in `s`.

In the former case (`rarray<float,3> s(256,256,256)`), calling `s.clear()` before the rarray goes out of scope would release the memory of this array.

The construction specifying a number of extents as arguments works for arrays with rank up to and including 11. For arrays with larger rank, you have to pass an pointer to the array of extents.

3.2 rvector, rmatrix, rtensor (c++11 only)

When compiling in c++11 mode, there are short cut types for one-dimensional, two dimensional and three dimensional arrays, called rvector, rmatrix and rtensor, respectively. The following equivalences hold:

```
rvector<T> = rarray<T,1>
rmatrix<T> = rarray<T,2>
rtensor<T> = rarray<T,3>
```

This only works in the c++11 standard and above.

3.3 Accessing the elements

The elements of rarray objects are accessed using the repeated square bracket notation as for automatic C++ arrays. Thus, if `s` is a `rarray` of rank `R`, the elements are accessed using `R` times an index of the form `[`n_i`]`, i.e. `s[`n_0`][`n_1`]...[`n_{R-1}`]` For example:

```
for (int i=0;i<s.extent(0);i++)
  for (int j=0;j<s.extent(1);j++)
    for (int k=0;k<s.extent(2);k++)
      s[i][j][k] = i+j+k;
```

In addition to the explicit assignment of each element, there is an alternative way to assign values to the elements of an rarray using a comma separate form, as follows

```
rarray<double,2> matrix(3,3);
matrix = 1,2,3,
         4,5,6,
         7,8,9;
```

This is usefully mostly for small matrices (as in tests), as entering large rarrays in this way is rather impractical.

3.4 Copying and function arguments

In C++, when we copy a variable of a built-in type to a new variable, the new copy is completely independent of the old variable. Likewise, the default way of passing arguments to a function involves a complete copy for built-in types. For C-style arrays, however, only the pointer to the first element gets copied, not the whole array. The latter is called a shallow copy. Rarrays use shallow copies much like pointers.

What does this essentially mean? Well:

1. You can pass rarrays by value to function, which is as if you were passing a pointer. (Passing by reference also works.)

2. When you assign one rarray to another, the other simply points to the old one.

3. If you wish to do a deep copy, i.e., create a new array independent of the old array, you need to use the `copy` method.

3.5 Returning a rarray from a function

Because rarray uses reference counting, returning a rarray from a function poses no problems.

Consider the function `zeros` used in `main()`:

```
#include <rarray>
rarray<double,2> zeros(int n, int m) {
   rarray<double,2> r(n,m);
   r.fill(0.0);
   return r;
}
int main() {
   rarray<double,2> s = zeros(100,100);
   return s[99][99];
}
```

In line 2, a rarray `r` is created, and filled, on line 3, with zeros. What happens on line 4, depends on whether the compiler supports the C++11 standard. If it does, things work as intended, in that on line 4, `r` gets moved out of the function and into `s`, using C++11's move semantics. Move semantics cause `r` to be left in an empty state that will not deallocate the memory associated with the array upon exiting the function, leaving that task to `s` in the caller. However, if the compiler does not support move semantices, then on line 4, the array `r` is shallowly copied out of the function and into `s` on line 7. Once copied into `s`, Because in this process, reference counters keep track of whether the data in the rarray is still used, `r` does not get destroyed or left in an invalid state, and things still work as intended.

3.6 Reshaping the rarray

To use the data in an rarray but access it in a different 'shape', one can

1. create a new rarray which uses the data from the first rarray. E.g.

```
void dump(const rarray<double,3>& r) {
   rarray<double,1> rflat(r.data(), r.size());
   for (int i=0;i<r.size();i++)
       std::cout << rflat[i] << ' ';
  std::cout << std::endl;
}
```

2. or one can reshape the existing rarray with the desired dimensions. E.g.

```
void dump(rarray<double,3> r) {
   r.reshape(r.data(), 1, 1, r.size());
   for (int i=0;i<r.size();i++)
       std::cout << r[0][0][i] << ' ';
  std::cout << std::endl;
}
```

The latter only works if the new and old shape have the same ranks.

It should be stressed that the last example has no side effects for the shape of the original rarray that was passed to the `dump` function, that original arrays retains its shape. Note that in both methods, the total size of the new shape should be less or equal to that of the old shape, because no additional data is allocated by the reshape method.

3.7 Optional bounds checking

If the preprocessor constant `RA_BOUNDSCHECK` is defined, an `out_of_bounds` exception is thrown if

- an index is too small or too large;
- the size of dimension is requested that does not exist (in a call to `extent(int i)`);
- a constructor is called with a zero pointer for the buffer or for the dimensions array;
- a constructor is called with too few or too many arguments (for `R`$<= 11$).

`RA_BOUNDSCHECK` can be defined by adding the `-DRA_BOUNDSCHECK` argument to the compilation command, or by `#define RA_BOUNDSCHECK` before the `#include <rarray>` in the source.

4 Comparison with standard alternatives

The more-or-less equivalent automatic array version

```
float arr[256][256][256];
arr[1][2][3] = 105;
```

is a little simpler, but automatic arrays are allocated on the stack, which is typically of limited size. Another drawback is that this array cannot be passed to functions that do not hard-code exactly matching dimensions except for the last one.

Using rarray also has benefits over another common C++ solution using the STL:

```
#include <vector>
int main() {
   using std::vector;
   int n = 256;                // size per dimension
```

```
    vector<vector<vector<float>>> v(n);// allocate for top dimension
    for (int i=0;i<n;i++) {
       v[i].resize(n);         // allocate vectors for middle dimension
       for (int j=0;j<n;j++)
          v[i][j].resize(n);   // allocate elements in last dimension
    }
    v[1][2][3] = 105;           // assign to element (for example)
 }
```

which is complicated, is non-contiguous in memory, and likely slower.

5 Class definition

5.1 Interface

Effectively, the interface part of a rarray object is defined as follows:

```
template<typename T, int R>
class rarray {
public:
 rarray();                                  // create uninitialized array
 rarray(int extent0, ...);                  // constructor for R<=11
 rarray(const int* extents);                // alternative: extents in array
 rarray(T* data, int extent0, ...);         // construct with existing data
 rarray(T* data, const int* extents);       // alternative: extents in array
 rarray(const rarray<T,R> &a);              // shallow copy constructor
 ~rarray();                                 // destructor
 void reshape(int extent0, ...);            // change copy's shape (not data)
 void reshape(const int* extent);           // change copy's  shape (not data)
 void         clear();                      // release memory
 void         fill(const T& value);         // fill with uniform value
 bool         is_clear();         const;    // check if uninitialized;
 rarray<T,R>  copy()              const;    // deep copy
 const int*   shape()             const;    // all extents as C-style array
 int          extent(int i)       const;    // extent in dimension i
 int          size()              const;    // total number of elements
 T*           data()              const;    // start of internal buffer
 T*const*...  ptr_array();        const;    // convert to a T*const*...
 T**...       noconst_ptr_array() const;    // converts to a T**...
 rarray<const T,R>&  const_ref()  const;    // convert to const elements
 rarray<T,R>& operator=(const rarray<T,R> &a);// shallow assignment
 rarray<const T,R-1> operator[](int i) const; // element access
 rarray<T,R-1>       operator[](int i);       // element access
};
```

5.2 Methods

Definition: `int extent(int i) const;`

This method returns the size of dimension `i`.

Definition: `const int* shape() const;`

This method returns the size of all dimensions as a c array.

Definition: `int size() const;`

This method returns the total number of elements.

Definition: `T* data() const;`

This method returns a pointer to the first element of the array.

Definition: `T*const*... ptr_array() const;`

This method returns a pointer to a `T*const*...`.

Definition: `T**... noconst_ptr_array() const;`

This method returns a pointer to a `T**...`. (This conversion breaks const-correctness.)

Definition: `rarray<const T,R>& const_ref() const;`

This method creates reference to this with const elements.

Definition: `void reshape(int extent0, ...);`

This method can be used to change the shape of the rarray. The data in the underlying buffer will not change. Note that the number of dimensions must remain the same and the total new size must be less or equal to the old size. This works only for arrays with rank up to and including 11.

Definition: `void reshape(const int* extents);`

Same as the previous `reshape` method, but takes an array of new dimensions as an argument. This is the only way to reshape an array of rank twelve or higher.

Definition: `void clear();`

This method release memory the memory associated with the rarray. If the rarray was created by providing a pre-existing buffer for the data, the memory of this buffer does not released.

Definition: `void fill(const T& value);`

This method fills the array with a uniform value `value`.

Definition: `bool is_clear();`

Checks if the rarray is in an uninitialized state.

Definition: `rarray<T,R> copy() const;`

This method creates deep copy of the rarray, i.e., an independent rarray with the same dimensions are the original rarray and with its content a copy of the original rarray's content.

6 I/O

6.1 Streaming input and output

Although it is usually preferrable to store large arrays in binary, rarray does provide streaming operators << and >> to read and write to standard C++ iostreams. You need to include an additional

header file called `rarrayio` for this.

The output produced by the output streaming operator is like that of automatic arrays initializers: Each dimension is started and ended by { and } and components are comma separated. Except for the inner dimension, newlines are included in the output, but no spaces, and no trailing newline. E.g.:

```
#include <iostream>
#include <rarray>
#include <rarrayio>
int main() {
  int buf[6] = {1,2,3,4,5,6};
  rarray<int,2> arr(buf,3,2);
  std::cout << arr;
}
```

will print the following:

```
{
{1,2},
{3,4},
{5,6}
}
```

Apart from inserting curly braces, commas, and newlines, The streaming operators use those of the element types.

The streaming operators are designed such that the format that is written out by the << operator, should be readable by the >> operator. Without further formatting tules, reading would not be unambiguous for some types, e.g. for `std::string`, that can contain the syntactic elements '{', '}', ',', or '#'. If these elements are found in the output of an element, the element is prepended with a string that encodes the string length of the output. This prepending string starts with a '#' character, then the length of the string output for the element (excluding the prepended part), followed by a ':' character.

An exception to the '#' formatting rule exists for types that stream out such that the first output character is '(' and the last is ')', with no other ')' characters intervening. Such output does not require the '#' formatting. Complex numbers are a prime example of a type with such output and input.

Note that while the input stream operators expect the same format as the output produces by the output streams, they but will not care if newlines are not there. However, there should not be any extraneous whitespace in the input.

7 Performance

7.1 General performance consideration

(Note: flags in this section are given in their linux notation).

`rarray` is written specifically with performance in mind. However, the way it manages to mimic the natural c and c++ way of using multi-dimensional arrays, i.e. with (repeated) square brackets, would come at a high cost of function calls and temporary objects. The rarray library tries to inline most function calls to avoid this cost, but it needs the c++ compiler's optimization capabilities to help out. Even the lowest optimization levels, such as obtained with the `-O1` flag, will enable this.

Rarray is meant to be used with higher levels of optimization. When compiled with higher optimization levels, such as `-O3 -fstrict-aliasing` for `g++`, inlining should make the rarray objects at least as fast as the textbook solution for multidimensional arrays mentioned in the introduction, for most compilers (tested with gcc 4.6.1 – 4.8.2 and, icc 12.1.5 – 13.1.1 on x86, and with xlC 12.1 on power 6).

7.2 Debugging

When debugging a code with a symbolic debugger (e.g. gdb), in addition to compiling with an option to include symbols[1] into the executatble (e.g. `-g`), one usually switches off the compiler's optimization capabilities (`-O0`), because it can change the order of instructions in the code, which makes debugging very difficult. As explained above, when working with rarray, turning of the compiler's optimization options, can cause the program to become quite a bit slower, which can also hamper debugging. If the bug is unlikely to occur in rarray, it may be worth trying build with a mild optimization level such as `-O1` for debugging.

If, on the other hand, the bug is in rarray, one likely needs to switch off optimization to see what is going on. In addition to the `-O0` flag, this also requires switching off forced inlining, using the `-DRA_INLINE=inline` flag.

7.3 Profiling

Sampling profiling tools (such as `gprof`) work by periodically recording the state and call stack of the program as it runs. They can be very useful for performance analysis. When using these tools with rarray, it is advisable to compile with a minimum level of optimization `-O1`. Otherwise, a lot of the internal function calls of rarray that could simply be optimized away, and would pollute the sampling.

A hint regarding the current state of gprof and gcc (Mar 2016): the newer gcc compilers encode symbols differently than earlier versions, and gprof relies on the earlier format. This can impede e.g. profiling by line number. Compiling (and linking) with the `-gstabs` flags enables the earlier way of encoding symbols in the application, and allows for gprof to function fully.

7.4 Memory overhead using the rarray class

The memory overhead here comes from having to store the dimensions and a pointer-to-pointer structure. The latter account for most of the memory overhead. A rarray object of $100\times100\times100\times100$ doubles on a 64-bit machine will have a memory overhead of a bit over 1%. In general, the memory

[1] Names of functions and other information regarding the code such as line numbers and file names are collectively called "symbols".

overhead as a percentage is roughly 100% divided by the last dimension. Therefore, avoid rarrays with a small last dimension such as $100\times100\times100\times2$.

7.5 Compilation overhead using the rarray class

There is an overhead in the compilation stage, but this is very compiler dependent.

7.6 More details regarding the performance overhead

As mentioned above, when compiled with optimization on (`-O3 -fstrict-aliasing` is a good default), inlining should make the rarray objects at least as fast as the textbook solution above, for most compilers (tested with gcc 4.6.1 – 4.8.2 and, icc 12.1.5 – 13.1.1 on x86, and with xlC 12.1 on power 6). So in that sense, no there is no performance overhead. However, with `RA_BOUNDSCHECK` defined, one incurs a compiler and machine dependent performance hit.

Let us give a bit more details as to why there should not be any performance overhead. Temporary intermediate expression in a multi-bracketed expression (such as a[5] in a[5][6][7]) are represented by an intermediate class called rarray_intermediate. This prevents assignment to such expressions and allows the optional bound checking to work. Most compilers (gcc, intel) can optimize these intermediate classes away when bounds checking is off. However some compilers (e.g. pgi) cannot do this and suffer from speed degredation because of these intermediate objects.

To get full speedup for these less-optimizing compilers, you can define the preprocessor constant `RA_SKIPINTERMEDIATE` (usually with a `-DRA_SKIPINTERMEDIATE` compiler option). Intermediate expression in a multi-bracketed expression are then replaced by bare pointers-to-pointers.

Note that when `RA_BOUNDSCHECK` is set, it will switch off `RA_SKIPINTERMEDIATE`.

8 Working with libraries

Performance was one reason for writing the rarray library. Being able to pass such arrays to numerical libraries was another.

Many numerical libraries, such as BLAS, LAPACK and FFTW, are C or fortran based and their API requires that

1. the array elements are stored contiguously in memory,
2. one passes the pointer to the first element, and
3. one passes the dimensions of the array explicitly.

Rarray helps in satisfying these requirement as follows:

1. Elements of a rarray are always stored contiguously in memory
2. The pointer to the first elements of the array can be found with the member function `data`
3. The dimensions of the arrays can be found from the`extent` member functions of rarray (or, for one-dimensional arrays, with the `size` function.

Here is an example to call the matrix-matrix multiplication routine from blas, using the cblas interface:

```
#include <iostream>
#include <rarray>
#include <rarrayio>
#include <cblas.h>
int main() {
   int m = 2, k=3, n =4;
   rarray<double,2> a(m,k), b(k,n), c(m,n);
   a = 1,-2,3,
       2,-1,0;
   b = -1,3,-2,1,
       -2,1,-3,2,
       -3,2,-1,3;
   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
               c.extent(0),c.extent(1),a.extent(1), 1.0,
               a.data(), a.extent(1),
               b.data(), b.extent(1),
               1.0,
               c.data(), c.extent(1));
   std::cout << "a=" << a << '\n'
             << "a=" << b << '\n'
             << "c=a*b=" << c << '\n';
}
```

See the blas documentation, for the precise meaning and operation of (c)blas.

Especially when working with complex data, an extra conversion may be required. For instance, working with the library called FFTW3, the following is a way to compute the fourier transform of a complex rarray:

```
#include <iostream>
#include <complex>
#include <rarray>
#include <rarrayio>
#include <fftw3.h>
int main() {
   int n = 4;
   rarray<std::complex<double>,1> a(n), b(n);
   a = 1.3+5.0i, 1.0, 5.0i, 2.0-5.0i;
   fftw_plan plan = fftw_plan_dft_1d(a.size(),
                    (fftw_complex*)a.data(), (fftw_complex*)b.data(),
                    FFTW_FORWARD, FFTW_ESTIMATE);
   fftw_execute(plan);
   fftw_destroy_plan(plan);
```

```
    std::cout << "a=" << a << '\n' << "FT(a)=" << b << '\n';
}
```

Note that this example uses the concise c++14 notation for complex variables (compile with `-std=c++14` or equivalent). Again, for the precise meaning of the fftw function call, see the documentation of that library.

Finally, it is worth noting that many numerical libraries care about the "alignment" of the data. Rarray currently does not have any way of facilitating alignment, but a user can allocate aligned data and pass it as a pre-existing buffer to the rarray constructor (see 3.1).

9 Conversions

9.1 Converting automatic C-style arrays to rarrays

It is possible to convert C-style automatic arrays to rarrays using the `RARRAY` macro (which calls some templated functions under the hood). You can apply `RARRAY` to any automatic array of rank six or less. The main convenience of this is that one can write functions that take rarray argument(s) and pass automatic arrays to them. Another use is in initializing a rarray. For example:

```
#include <iostream>
#include <rarray>
void print2d(const rarray<float,2> &s) {
    for (int i=0; i<s.extent(0); i++) {
        for (int j=0; j<s.extent(1); j++)
            std::cout << s[i][j] << ' ';
        std::cout << std::endl;
    }
}
int main() {
    float printme[4][4] = { { 1.0, 1.2, 1.4, 1.6},
                            { 2.0, 2.2, 2.4, 2.6},
                            { 3.0, 3.2, 3.4, 3.6},
                            { 4.0, 4.2, 4.4, 4.6} };
    print2d(RARRAY(printme));
    rarray<float,2> a = RARRAY(printme).copy();
    print2d(a);
}
```

9.2 Conversions for function arguments

A function might take a `rarray<const T,R>` parameter if elements are not changed by it. Because C++ cannot convert a reference to a `rarray<T,R>` to a `rarray<const T,R>`, one has to use the `const_ref` method to do this for you. For example:

```
float add(const rarray<const float,2> &s)   {
   float x = 0.0;
   for (int i=0; i<s.extent(0); i++)
      for (int j=0; j<s.extent(1); j++)
         x += s[i][j];
   return x;
}
int main() {
   rarray<float,2> s(40, 40);
   float z = add(s.const_ref()); // because add() takes <const float>
}
```

rarray objects are also easy to pass to function that do not use rarrays. Because there are, by design, no automatic conversions of a rarray, this is done using methods.

There are two main ways that such functions expect a multidimensional array to be passed: either as a pointer (a T*) to the first element of the internal buffer composed of all elements, or as a pointer-to-pointer structure (a T**...). In the former case, it may be important to know that a rarray stores elements in row-major format.

With the const keyword, the number of useful C++ forms for multidimensional array arguments has grown to about six. In the case of a two-dimensional array these take the forms: T*, const T*, T*const*, const T*const*, T**, and const T**. Using the rarray library, const-correct argument passing requires the data or ptr_array method but non-const-correct argument passing will require the noconst_ptr_array function, possibly combined with const_ref. We will briefly looking at these cases separately now.

9.3 Conversion to a T* or a const T*

A function may expect a multidimensional array to be passed as a simple pointer to the first element, of the form T*, or of the form const T*. This is the case for most c or fortran libraries, as disussed above in 8. A rarray object s of type rarray<T,I> can be passed to these functions using the syntax s.data(), which yields a T*.

Example 1:

```
void fill1(float* a, int n1, int n2, float z) {
   for (int i=0; i<n1*n2; i++)
     a[i] = z;
}
int main() {
   rarray<float,2> s(40, 40);
   fill1(s.data(), s.extent(0), s.extent(1), 3.14);
}
```

Example 2:

```
float add2(const float* a, int n1, int n2) {
   float x = 0.0;
   for (int i=0; i<n1*n2; i++)
       x += a[i];
   return x;
}
int main() {
   rarray<float,2> s(40, 40);
   float z = add2(s.data(), s.extent(0), s.extent(1));
}
```

C++ accepts a `float*` instead of a `const float*`, so `data()` can be used in the latter example.

9.4 Conversion to a `T*const*` or a `const T*const*`

In `T*const*`, the middle const means that one cannot reassign the row pointers. The rarray classes can be converted to this type using the `ptr_array()` method. For higher dimensions, this case generalizes to `T*const*const*`, `T*const*const*const*`, etc.

Example 1:

```
void fill3(float*const* a, int n1, int n2, float z) {
   for (int i=0; i<n1; i++)
      for (int j=0; j<n2; j++)
         a[i][j] = z;
}
int main() {
   rarray<float,2> s(40, 40);
   fill3(s.ptr_array(), s.extent(0), s.extent(1), 3.14);
}
```

Example 2:

```
float add4(const float*const* a, int n1, int n2) {
   float x = 0.0;
   for (int i=0; i<n1; i++)
      for (int j=0; j<n2; j++)
         x += a[i][j];
   return x;
}
int main() {
   rarray<float,2> s(40, 40);
   float z = add4(s.ptr_array(), 40, 40);
}
```

C++ accepts a `T*const*` where a `const T*const*` is expected, so here one can again use the method `ptr_array()`. This extends to its generalizations `const T*const*const*`, `const T*const*const*const*`, etc., as well.

9.5 Conversion to a `T**`

Generating a T** from a rarray object, one could change the internal structure of that rarray object through the double pointer. This is therefore considered a not const-correct. It is however commonly needed, so `rarray` does have a function for it, called `noconst_ptr_array`.

Example:

```
void fill5(float** a, int n1, int n2, float z) {
   for (int i=0; i<n1; i++)
      for (int j=0; j<n2; j++)
         a[i][j] = z;
}
int main() {
   rarray<float,2> s(40, 40);
   fill5(s.noconst_ptr_array(), s.extent(0), s.extent(1), 3.14);
}
```

9.6 Conversion to a `const T**`

C++ does not allow conversion from T** to `const T**`. To convert to a `const T**`, one first needs to convert the `rarray<T,R>` to a `rarray<const T,R>` using `const_ref()`, after which one can use the `noconst_ptr_array` function.

Example:

```
float add6(const float** a, int n1, int n2) {
   float x = 0.0;
   for (int i=0; i<n1; i++)
      for (int j=0; j<n2; j++)
         x += a[i][j];
   return x;
}
int main() {
   rarray<float,2> s(40, 40);
   float z = add6(s.const_ref().noconst_ptr_array(), 40, 40);
}
```

A Installation

To use this header-only library, you can copy the files rarray, rarrayio, rarraymacros.h, rarraydelmacros.h to the directory ”/usr/include” or some other directory that the compiler searches for header files. Rather than doing so by hand, you can use the included Makefile, which can also compile the unit tests and benchmarks. To install without running tests or benchmarks, simply do:

```
make install PREFIX="[BASEDIR]"
```

to install the header files in the ”[BASEDIR]/include” directory, and the documentation in ”[BASEDIR]/share/doc”. If the include directory is not in the compiler’s search path, you will need to pass an option to the compiler directing to that directory (i.e. -I[BASEDIR]/include). If you have sudo permissions, you can also do

```
sudo make install
```

which is the same as `sudo make install PREFIX=/usr`.

For the tests and benchmarks, one first needs to run

```
./configure
```

to set up the makefile for your compiler. Note that to pick your compiler, you should have the CXX environment variable point to the right compiler command. If you have a recent gnu, intel, ibm or clang compiler, and the ’make’ command, then the Makefile should be enough to compile and run the unit tests and benchmarks:

```
make test
make valgrindtest
make benchmark
```

Be aware that the Makefile has not been extensively tested.

The rarray library has been tested with the GNU g++ compiler version 4.8.0 and up, the Intel C++ compiler version 13, IBM’s XL C++ compiler version 12 and up, and clang 3.5.