

rarray: Reference-Counted Multidimensional Arrays for C++

Ramses van Zon

November, 2023 (version 2.6.1)

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

What:

Reference-counted and non-owning multidimensional arrays with runtime dimensions.

What not:

No strides, no linear algebra, overloaded operators etc.

Why:

Usually faster than alternatives.

Uses the same accessors as automatic arrays.

Requires only the C++11 standard.

Data is contiguous to allow interfacing with libraries like BLAS, LAPACK, FFTW, etc.

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. Streaming I/O is also supported.

Define a $n \times m \times k$ array of floats:	<code>rarray<float,3> b(n,m,k);</code>
Define a $n \times m \times k$ array of floats with data pre-allocated at a pointer <code>ptr</code> :	<code>rarray<float,3> c(ptr,n,m,k);</code>
Element i,j,k of the array <code>b</code> :	<code>b[i][j][k]</code>
Pointer to the contiguous data in <code>b</code> :	<code>b.data()</code>
Total number of elements in <code>b</code> :	<code>b.size()</code>
Extent in the i th dimension in <code>b</code> :	<code>b.extent(i)</code>
Array of all extents of <code>b</code>	<code>b.shape()</code>
Define an array with same shape as <code>b</code>	<code>rarray<float,3> b2(b.shape());</code>
Shallow, reference-counted copy of the array:	<code>rarray<float,3> d=b;</code>
Deep copy of the array:	<code>rarray<float,3> e=b.copy();</code>
A rarray using an existing automatic array:	<code>float f[10][20][8]={{{{...}}}};</code> <code>rarray<float,3> g(f);</code>
Output a rarray to console:	<code>std::cout << h << std::endl;</code>
Read a rarray from console:	<code>std::cin >> h;</code>

Contents

1	For the impatient: the what, why and how of rarray	1
2	Introduction	3
3	Using rarrays	6
4	Comparison with standard alternatives	9
5	Class definition	11
6	I/O	12
7	Utilities: xrange and linspace	13
8	Performance	15
9	Working with libraries	16
10	Conversions	18
A	Installation	23
B	History	24

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 and deallocation constructs in the operators `new` and `delete` in the standard. A standard way to allocate a one-dimensional array is as follows:

```
int n = 1000;
float* a = new float[n];
a[40] = 2.4; // as an example on how to use this, access element at offset 40
delete[] a;
```

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

This style of allocation with a “raw” pointer is discouraged in C++ in favor of using “smart” pointers, which is possible since the C++17 standard:

```
int n = 1000;
std::unique_ptr<float[]> a(new float[n]);
a[40] = 2.4; // as an example on how to use this, access element at offset 40
// a gets deallocated automatically, or one can explicitly call a.reset(nullptr)
```

Automatic deallocation happens when the variable `a` goes out of scope. A unique pointer cannot be copied. Instead of `unique_ptr` one can use `shared_ptr`, which can be copied and keeps a reference counter to know when to deallocate the memory. Then, deallocation happens when the life time of all copies of the `shared_ptr` has ended.

In the above code snippets, the `new` construct and the `std::unique_ptr`/`std::shared_ptr` assign the address of the array to a pointer. These pointers do not remember their size, so they are not really an ‘array’. The standard C++ library does provide a dynamically allocated one-dimensional array that remembers its size, in the form of the `std::vector`, e.g.

```
int n = 1000;
std::vector a(n);
a[40] = 2.4; // as an example on how to use this, access element at offset 40
// a gets automatically deallocated, or one can explicitly call a.clear()
```

Multi-dimensional runtime-allocated arrays are currently not supported yet by C++, but there is a proposal for a non-owning multidimensional array in the C++23 standard, and C++26 may have an owning multidimensional array.

To handle these kinds of arrays in C++, the (early) textbook solution for multidimensional arrays that are dynamically allocated during runtime, would be as follows (here for a three-dimensional array of `dim0×dim1×dim2`)

```
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];
}
A[1][2][3] = 105; // as an example
```

Apart from the fact this will soon be obsolete, drawbacks of this solution are:

- the elements are not stored contiguously in memory, making this multi-dimensional array unusable for many numerical libraries,
- one has to keep track of array dimensions, and pass them along to functions,
- the intermediate pointers are non-const, so the internal pointer structure can be changed whereas, conceptually, it ought to be of type `float*const*const*`.

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
- Nested vectors from the Standard Template Library; and
- Kokkos's reference implementation of the C++23 `mdspan` template.

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 of arbitrary rank with *minimal to no performance overhead* compared to the textbook solution. From the above list of library solutions, only the implementation in Kokkos of the non-owning `mdspan` has virtually no overhead.

`Rarray` aims to be simpler; all it takes to create a three-dimensional **Example**:

```
#include <rarray>
...
rtensor<float> A(dim0,dim1,dim2);
A[1][2][3] = 105; // as an example
...
```

Design Points of `rarray`

1. To have dynamically allocated multidimensional arrays that combine the ease and convenience of automatic C++ arrays while being compatible with the typical textbook-style dynamically allocated pointer-to-pointer structure.

The compatibility requirement with pointer-to-pointer structures is achieved by allocating a pointer-to-pointer structure. This structure accounts for most of the memory overhead from using `rarray`.

2. To be as fast as pointer-to-pointer structures.
3. To do shallow copy by default, deep copy possible.
4. To have `rarrays` know their sizes, so that they can be passed to functions as a single argument.
5. To enable interfacing with libraries such as BLAS, LAPACK and FFTW: this is achieved by guaranteeing contiguous elements in the multi-dimensional array, and a way to get this data out.

The guarantee of contiguity means strided arrays are not supported.

6. To allow sharing between components of an application while avoiding dangling references. This is possible by utilizing reference counting.
7. To allow `rarrays` to hold non-owning views that use an existing buffer, without having to use a separate type.
8. To avoid some of the cluttered semantics around `const` correctness when converting to pointer-to-pointer structures when interfacing with legacy code.

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, (atomic) reference counted, copies by default, but also has a deep copy method.
- Use of C++11 move semantics 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`.
- A method `empty` 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`). Used in automatic conversion from `rarray<T,R>` to `rarray<const T,R>`.
- 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.
- Checks index bounds if the preprocessor constant `RA_BOUNDSCHECK` is defined.

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);  
// do whatever you need with s  
s[1][2][3] = 105;
```

In this case, calling `s.clear()` would release the memory of this array. Otherwise, the memory will be released when the rarray goes out of scope. That is unless a copy of `s` was made before either of these happened.

It is also possible to use an external, pre-allocated buffer, as follows:

```
std::unique_ptr<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
```

Note that `s` will have dangling references (often leading to “Segmentation faults”) if `pre_alloc_data` is deallocated while `s` is not gone out of scope or `clear`-ed.

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 Shorthand rarray types: `rvector`, `rmatrix`, `rtensor`

For convenience, rarray defines shortcut 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>
```

for any type `T`.

3.3 Accessing elements of an rarray

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 `[ni]`, i.e. `s[n0][n1]...[nR-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, for instance, for small matrices (e.g. in tests). Entering large rarrays in this way would be 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, but use atomic reference counting to know when memory can be released (similar to the `std::shared_ptr<T>` of C++11 and `std::shared_ptr<T[]>` of C++14).

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 implements move semantics, returning an rarray from a function does not pose any 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 3, a rarray `r` is created, and filled, on line 4, with zeros. On line 5, `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 `s`.

3.6 Reshaping the rarray

To use the data in an rarray but access it in a different 'shape', one has several options.

1. One can 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. Alternatively, one can reshape the existing rarray with the desired dimensions. E.g.

```
void dump(rarray<double,3> r) {
    r.reshape(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 and the number of elements stays the same. It is possible to reshape an rarray to have less elements, but only by explicitly passing a last parameter `ra::RESIZE::ALLOWED`, e.g.


```
void dump(rarray<double,3> r) {
    r.reshape(1, 1, r.size()/2, ra::RESIZE::ALLOWED);
    for (int i=0;i<r.size();i++)
        std::cout << r[0][0][i] << ' ';
    std::cout << std::endl;
}
```

It should be stressed that the last examples have no side effects for the shape of the original rarray that was passed to the function. The rarray was passed by value, which means the function holds a shallow copy, but the reshape method creates a new independent shape while using the same buffer as the original array. The original array retains its shape. If the array was passed by reference, the original array would be changed when applying reshape.

As a design decision, the reshape method should not allocate any memory. Thus, it is never possible to do a reshape that leads to more elements.

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 shape array;

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

Compared to the old textbook method of declaring an array (see above), or the rarray method:

```
#include <rarray>
int main() {
    int n = 256;
    rarray<float,3> arr(n,n,n);
    arr[1][2][3] = 105; // for example
}
```

the more-or-less equivalent automatic array version

```
int main() {
    int n = 256;
    float arr[n][n][n];
    arr[1][2][3] = 105; // for example
}
```

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

Using rarray also has benefits over another C++ solution, i.e. using the `std::vector` class from the Standard Template Library:

```
#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.

C++23 will have a non-owning library, `mdspan`, which should work roughly as follows:

```
#include <memory>
#include <mdspan>
int main() {
    int n = 256;                // size per dimension
    std::unique_ptr<float[]> p (new float[n*n*n]); // or vector or a shared_ptr
    using exts = std::extents<size_t, std::dynamic_extent,
        std::dynamic_extent, std::dynamic_extent>;
    std::mdspan<float, exts> (vector.data(), exts(n,n,n));
    v[1,2,3] = 105;             // assign to element (for example)
}
```

Which is more involved than the rarray solution and does not offer reference counting.

5 Class definition

5.1 Interface

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

```
namespace ra {
template<typename T, rank_type R>
class rarray {
public:
    rarray(); // create uninitialized array
    rarray(size_type extent0, ...); // constructor for R<=11
    rarray(const size_type* extents); // alternative: extents in array
    rarray(T* data, size_type extent0, ...); // construct with existing data
    rarray(T* data, const size_type* extents); // alternative: extents in array
    rarray(const rarray<T,R> &a); // shallow copy constructor
    ~rarray(); // destructor
    void reshape(size_type extent0, ...); // change shape (not data) for R<=11
    void reshape(const size_type* size_type); // change shape (not data)
    void clear(); // release memory
    void fill(const T& value); // fill with uniform value
    bool empty() const; // check if uninitialized
    rarray<T,R> copy() const; // deep copy
    const size_type* shape() const; // all extents as C-style array
    size_type extent(int i) const; // extent in dimension i
    size_type size() const; // total number of elements
    rank_type rank() const; // rank, i.e, the value of R
    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
    operator[](size_type i) const; // enables const element access
    operator[](size_type i); // enables element access for assignment
    rarray<T,R-1> at(size_t i); // retrieve the ith 'row' with bounds checking
};
}
```

5.2 Methods

Definition: `size_type extent(rank_type i) const;`

This method returns the size of dimension `i`.

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

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

Definition: `size_type size() const;`

This method returns the total number of elements.

Definition: `rank_type rank() const;`

This method returns the number of dimensions.

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*...`. Note that rarrays have an automatic conversion to this type.

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(size_type 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 (i.e., the rank) 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 empty();`

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 preferable to store large arrays in binary, rarray does provide streaming operators `<<` and `>>` to read and write to standard C++ iostreams in text format.

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>
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 the streaming operators 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 rules, 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 primary 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 Utilities: xrange and linspace

The rarray library contains a few more utilities.

7.1 xrange

The xrange function returns an iterable generator, which can produce numbers from a starting point, with some step size, up to but not including an end point. In the simplest case, this can be used as a counter, e.g.:

```
#include <iostream>
#include <rarray>
int main() {
    for (int i: xrange(1000))
        std::cout << i << " ";
    std::cout << std::endl;
}
```

will print the numbers 0 to 999, which are generated in succession, i.e., no integer array of size 1000 containing these numbers is ever created. The return type of the xrange function is an `ra::Xrange<T>` object, with `T` equal to `int` in the above example. This is an iterable object that can be used in a range-based for. It could also be used as the source of a copy, and thus to initialize an rarray, e.g. you could get an array with values 0,2,4,...,998, as follows

```
#include <rarray>
#include <algorithm>
int main() {
    ra::Xrange<int> in = xrange(0,1000,2);
    rvector<int> r(500);
    std::copy(in.begin(), in.end(), r.begin());
    ...
}
```

(replacing the type `ra::Xrange<int>` with `c++11`'s `auto` seems like a good idea here).

The general form of the range function is

```
ra::Xrange<T> xrange(T endvalue)
ra::Xrange<T> xrange(T beginvalue, T endvalue, T stepsize=1)
```

In the first form, the begin value is 0 and the stepsize is 1. These are template functions, where `T` can be any type that can be converted to a double, although the most useful cases are those where `T` is an integer. In all cases, the first generated value is exactly `beginvalue`, each subsequent value is one stepsize larger, and the `endvalue` is the first value that is not generated (to be precise, the number of numbers generated is `ceil((beginvalue-endvalue)/stepsize)`, using floating point division).

The case where `endvalue` is less than `beginvalue`, or for which `stepsize` is negative, are, in version 2.1 - 2.6 of rarray, undefined.

7.2 linspace

The `linspace` function returns a rvector with a specified number of points between two given values, inclusively. The general form of the function is

```
rvector<T> linspace(T x1, T x2, int n, bool end_incl=true);
```

Here, `x1` is the first value, `x2` the last value, and `n` is the number of values. If the latter is not given or is set to zero, the number of values is such that the stepsize is as close to one as possible. If `end_incl` is set to false, the generated values are such as if the number of points is `n+1` but the last value is omitted.

For example, to create an `rvector` with 101 equally spaced values between `-1.0` and `1.0`, one would do

```
#include <rarray>
int main() {
    rvector<double> r = linspace(-1.0, 1.0, 101);
    ...
}
```

The first argument of `linspace` is allowed to be greater than the last, in which case, decreasing values are generated. The two arguments are allowed to be equal as well, which generates a vector with all equal values. In that case, `end_incl` can not be set to false. The case where the number of points is 1 and `end_incl=false` is ill defined.

Note that for integer types, using `linspace` without specifying their number (i.e. `linspace(n1,n2)`) gives the same values as are generated by the `xrange` function without a stepsize and with the end-value one higher (i.e., `xrange(n1,n2+1)`).

8 Performance

8.1 General performance consideration

`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 for many compilers, though some will need high levels of optimization.

`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.

8.2 Debugging

When debugging a code with a symbolic debugger (e.g. `gdb`), in addition to compiling with an option to include "symbols" (i.e., names of functions and other information regarding the code) into the executable (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 off 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.

8.3 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.

8.4 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 $100 \times 100 \times 100 \times 100$ doubles on a 64-bit machine will have a memory overhead of a bit over 1%. In general, the memory overhead as a percentage is roughly 100% divided by the last dimension. Therefore, avoid rarrays with a small last dimension such as $100 \times 100 \times 100 \times 2$.

8.5 Compilation overhead using the rarray class

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

9 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 or shape member functions of rarray (or, for one-dimensional arrays, also with the size member function).

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

```
#include <iostream>
#include <rarray>
#include <cblas.h>
int main() {
    int m=2, k=3, n=4;
    rmatrix<double> 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'
                << "B=" << B << '\n'
                << "C=A*B=" << C << '\n';
}
```

See the (C)BLAS documentation for the precise meaning and operation of cblas functions.

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 <fftw3.h>
int main() {
    int n = 4;
    rvector<std::complex<double>> 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).

10 Conversions

10.1 Converting automatic C-style arrays to rarrays

It is possible to convert C-style automatic arrays to rarrays if they have rank of at most 11. 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 stackarray[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} };

    // automatic conversion
    print2d(stackarray);
    // view using the same data
    auto a = rarray<float,2>(stackarray);
    print2d(a);
    // independent copy of the same data
    auto b = rarray<float,2>(stackarray).copy();
    print2d(b);
}
```

10.2 Conversions for function arguments

A function might take a `rarray<const T,R>` parameter if elements are not changed by it. C++ cannot automatically convert a template with a type `T` to one with a type `const T`, ie., from `rarray<T,R>` to a `rarray<const T,R>`. However, `rarray` provides this feature nonetheless. 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); // note: not const!
    float z = add(s); // yet this works
}
```

One can also explicitly use the `const_ref` method to do this conversion.

`Rarray` objects are also easy to pass to functions 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.

10.3 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 discussed above in 9. 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.

10.4 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, but `rarrays` also an automatic conversion to this type. 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);
    // or, since rarray 2.0:
    fill3(s, 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);
    // or,, since rarray 2.0,  z = add4(s, 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.

10.5 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);
}

```

10.6 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

Rarray is a header-only library, so you could just copy the header `rarray` to the directory `/usr/include` or some other directory where your C++ compiler looks for header files. Alternatively, you could use the command

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

This installs the header file in the `[BASEDIR]/include` directory, and the documentation in `[BASEDIR]/share/doc`. If you have `sudo` permissions, you can also do

```
sudo make install
```

to install the header and documentation to `/usr/include` and `/usr/share/doc`, respectively. Note that this will fail on recent MacOS versions, in which case, try `sudo make install PREFIX=/usr/local`.

To modify `rarray`, do not edit the `rarray` header file, as this is a generated file. Instead, you should edit the files in the `headersources` directory. You can use the included Makefile to assemble the `rarray` headers with

```
make headers
```

If you're editing the documentation (this file), you need `pdflatex`, and you should edit `rarraydoc.tex` and then do

```
make doc
```

The Makefile can also be used to compile and run the unit tests and benchmarks. Simply do:

```
./configure  
make test  
make benchmarks
```

The `configure` command should work under Linux if you have a recent GNU, Intel, IBM or Clang compiler. Note that to pick your compiler, you may have to set the `CXX` environment variable point to the right compiler command (e.g. `export CXX=clang++`) before running `configure`.

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`) or, for `gcc`, `clang` and `intel` compilers, set the `CPATH` environment variable.

B History

Dec 2013:

First implementation of the header-only library `rarray` for runtime multidimensional arrays.

Jan 2014: Version 1.0

Code put on github

Mar 2015: Version 1.1

Changed the text output format used to use newlines

Feb 2017: Version 1.2

Added C++11 aliases `rvector`, `rmatrix` and `rtensor`. Eliminated the need for `rarraymacros.h` and `rarraydelmacros.h` in the installed version of `rarray`.

Oct 2019: Version 2.0

Prompted by a deeply rooted memory leak, this version is a full rewrite of `rarray` from the ground up, leaving most of the application interface unchanged. `Rarray` now **requires** C++11. The optional but expensive capability to do index range checking has partially been lost in the rewrite. `Rarray` comes with its own unit testing library now, `'rut'`.

Jan 2020: Version 2.1

Reference counting of `rarray` data is now done atomically, so copying `rarrays` should now be thread-safe. One-character names of substructures, used in debugging `rarray` itself, were removed. Streaming operators for `rarrays` moved to the `ra` namespace.

Feb 2020: Version 2.2

Several bug fixed with running and installing `rarray` on MacOS. The `rarray` unit test library `'rut'` was dropped in favour of `'catch2'`, which does everything that `'rut'` was intended to do, but better. Since `'catch2'` is header only, this makes running the tests on different platforms much easier.

Jan 2022: Version 2.3

Streaming I/O now part of the `rarray` header. `rarrayio` header is obsolete, but kept for backwards compatibility.

Dec 2022: Version 2.4

Internal refactoring focussed on eliminating warnings, dead code, version tracking in code, and exception safety. Fixed bug for compound data types. Support added for Intel OneAPI's icpx C++ compiler.

Dec 2022: Version 2.5

Optional bounds checking is back.

`is_clear` renamed to `empty`.

Can now get a subarray with square brackets.

`Rarray` objects no longer automatically convert into `T*const*...` pointers.

More extensive unit and coverage tests.

Nov 2023: Version 2.6

Bug fixes (inlining, compiler settings, exception safety). Better support for `rarray<const T,R>`. Implicit conversion operator from `rarray<T,R>` to `rarray<const T,R>`. Started support for the multidimensional subscript operator for C++23 compliant compilers. Code cleanup.