rarray: Multidimensional Runtime Arrays for C++

Ramses van Zon

June 5, 2015

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.

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 already 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 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;`

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`. An example of the standard way to allocate a one-dimensional array is the following piece of code:

```
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.

```
std::vector a;
int n = 1000;
a.reserve(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 cluttered sematics around const when converting to pointer-to-pointer structers.

Features of rarray:

- Can use any data type `T` and any rank `R`.
- Views on existing contiguous arrays.
- Does shallow copies by default, but also has a deep `copy` method.
- Elements are accessible using repeated square brackets, like C/C++ arrays.
- 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 the rarrays

3.1 Defining a multidimensional rarray

To use rarray, first include the header file rarray:

```
#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, already allocated buffer, as

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

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().

Even in the former case (rarray<float,3> s(256,256,256)), one can release the memory of this array by calling s.clear() before the rarray goes out of scope.

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 Accessing the elements

The elements of rarray objects are accessed using the repeated square bracket notation as for automatic C++ arrays (by design). 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;
```

3.3 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, with the additional functionality that memory allocated by the rarray gets released.

What does this essentially mean? Well:

1. You can pass rarrays by value to function, which is as if you were passing a pointer.
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.

3.4 Returning a rarray from a function

Unless you're using rarray in a C++11 context, the shallow copying of rarrays causes problems when returning a rarray from a function.

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`, `r` is destroyed and releases its memory, but this leaves `s` in an invalid state (as it points to memory that is no longer allocated).

To resolve this, when used in C++03, rarray can mimic this move behavior, but it requires a hand from the programmer. One can tell the function `zeros` in the example above to move the array `r` into a wrapper class of type `rarray<T,R>::return_type`, which when assigned to `s`, achieves the desired affect of making `s` responsible for memory allocation, and leaving `r` in an empty state. In the above example, it should be used as follows:

```
rarray<double,2>::return_type zeros(int n, int m) {
   rarray<double,2> r(n,m);
   r.fill(0.0);
   return r;
}
```

In C++11, this code will still work, as the `rarray` header file contains a typedef of `return_type` to `rarray`.

3.5 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;
    }
```

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

The last example has no side effects for the shape of the original rarray that was passed to the `dump` function 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.

By default, `reshape` will only succeed for shallow copies of rarrays, but fail for original arrays.[1] In the above example, `r` is an argument, and it is therefore a shallow copy of some other array, and thus can be reshaped. If the argument had been of reference type (e.g. `rarray<double,3>& r`), then the reshape would have failed.

This restriction can be annoying in contexts where you want to reshape an original array and you know no shallow copies exist. You can force reshape to do its work by using `reshape_force` instead of `reshape`. So the following would work

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

This is not a recommend technique: existing shallow copies of the rarray `r` would, as a side effect, become invalid after calling the function, without a way to detect it.

3.6 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.

[1] The motivation for this behavior is that reshaping an original array would make any exisiting shallow copies invalid.

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].reserve(n);          // allocate vectors for middle dimension
      for (int j=0;j<n;j++)
         v[i][j].reserve(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 reshape_force(int extent0, ...);   // change owner shape (not data)
 void reshape_force(const int* extent);  // change owner 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 a `T*`. Const-correct.

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

This method returns a pointer to a `T*const*...`. Const-correct.

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. Warning: if the rarray is not a shallow copy of another rarray, the resizing will **not** be performed, to protect any existing shallow copies (but see `reshape_force` below).

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 reshape_force(int extent0, ...);`

The `reshape` method fails if the rarray is an original rarray, to protect existing shallow copies. If no such copies exist, one can use the `reshape_force` method to reshape an original array.

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

Same as the previous `reshape_force` 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 Conversions

6.1 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, and even to a rarray (which essentially does nothing). 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);
}
```

6.2 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.

6.3 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*`. 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.

6.4 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.

6.5 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);
}
```

6.6 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);
}
```

7 I/O

7.1 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. 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 {{1,2},{3,4},{5,6}}. The streaming operators use those of the element types.

Reading would not be unambiguous for some types, e.g. for `std::string`, that can contain the syntactic elements '{', '}', ',', or '#'. If thse 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.

The input stream operators expect the same format as input.

8 Overheads

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

8.2 Performance overhead using the rarray class

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`.

8.3 Compilation overhead using the rarray class

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